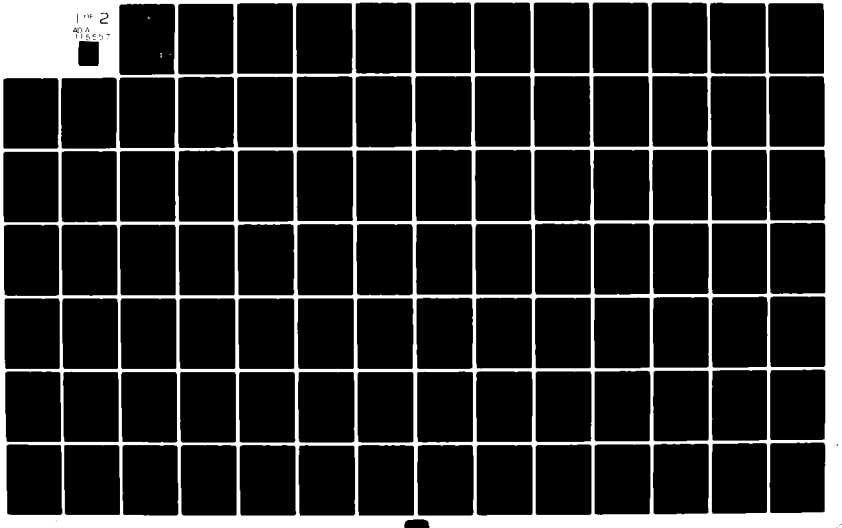
AD-A115 557 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
ARBITRARY PRECISION IN A PRELIMINARY MATH UNIT FOR ADA.(U)
MAR 82 P K LAWLIS
UNCLASSIFIED AFIT/GCS/MA/82M-2 NL



DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY (ATC)

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

This document has been approved for public release and sale; its distribution is unlimited.

AFIT/GCS/MA/82M-2

0

ARBITRARY PRECISION IN A PRELIMINARY MATH UNIT FOR ADA

THESIS

AFIT/GCS/MA/82M-2 Patricia K. Lawlis
Capt USAF

Approved for public release; distribution unlimited

AFIT/GCS/MA/82M-2

ARBITRARY PRECISION IN A PRELIMINARY MATH UNIT FOR ADA

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

by

Patricia K. Lawlis, B.S.

Capt USAF

Graduate Computer Systems

March 1982

Contents

Preface ii

List of Figures v

List of Tables vi

Abstract vii

I. Introduction 1

Background 2
Problem 5
Scope 6
Assumptions 7
General Approach 7
Sequence of Presentation 8

II. Project Design 10

The Structure 10
Top Level 10
Intermediate Levels 12
Low Levels 15
Algorithms 18
Number Representations 23

III. Software Implementation 27

Bit Manipulations and Implementation Independence 28
MATHPAK 32
IMPLMENT 38
Testing 43

IV. Conclusions and Recommendations 49

Completion 49
Use 51
Improvements and Enhancements 53

Bibliography 55

Appendix A: MATHPAK Structure 58

Appendix B: IMPLMENT Structure 60

Appendix C: MATHPAK Listing 61

Appendix D: IMPLMENT Listing 100

VITA 126

Preface

The project described in this paper was spawned in a very natural way. Only very recently has the development of the Ada computer language been completed. For this student of computer languages looking for a thesis project, Ada provided an obvious starting point. The only real problem was in pinpointing the particular way to use Ada in the thesis. However, since this writer is also a student of numerical methods, it was not hard to decide on a numeric application.

As with most projects, this thesis project changed with time. It began with a simple suggestion from an AFIT instructor, Capt Richard Conn. He mentioned that every language environment needs to include routines which perform standard math functions, and he suggested that developing such a unit of math functions for Ada would make a good thesis project.

After discussing Capt Conn's suggestion with another AFIT instructor, Capt Roie Black, the original thesis proposal took shape. We talked about developing a math unit which would be constructed as a "package" in Ada. He mentioned a preliminary Ada compiler project which was then being completed by another thesis student, Capt Alan (Ray) Garlington. But it would only implement integer type numbers. Capt Black suggested that adding a floating point capability to this compiler would permit the testing of the math unit during development. This thesis student

certainly wanted to be able to test what would be developed, so the suggestion was appealing. Hence, the original proposal was for a two-part effort: first, I would incorporate the real (floating point) type into Garlington's compiler, and then I would develop the math unit, testing it with the compiler.

However, after I got into my research, I began to see that the compiler effort would take a great deal away from the actual purpose for this thesis, and that no more than a small driver program would be required for the testing I would need to do. Hence, I decided to completely decouple this thesis project from Garlington's compiler. But this was in no way meant to take away from the compiler project. To be sure, the math unit is still intended to interface with the compiler later on, and both projects should eventually be integrated and become part of an AFIT Ada environment.

With both encouragement and helpful suggestions from Capt Black, I decided to make it possible for the precision of the math unit's calculations to be specified by the user, or, in other words, the unit would use arbitrary precision. This then became the driving force for the thesis project which emerged. The other important emphasis was to aim for the Ada goal of implementation independence. This was tricky in the Pascal implementation. However, I think I essentially achieved it in the end by having the implementor specify the values

for the constants to set the unit up for the environment.

I would like to express my appreciation to my thesis advisor, Capt Black, who devoted much of his valuable time toward this project, and who knew how to give just the right amount of advice. I would also like to give a special thanks to my good friend and colleague, Karyl Adams, whose interest and encouragement sustained me through the hard times.

Thanks also goes to the Wright-Patterson Avionics Laboratory, which sponsored this thesis project, and to the members of my committee, LtCol James Rutledge, Maj Michael Wirth (who has recently become a civilian), and Capt Conn, for all their help and all the time they gave to this project. To numerous friends and relatives who were good listeners when I needed to talk about this thesis work, even though they didn't really understand it, thanks a bunch.

My deepest gratitude goes to my husband, Mark, and my six year old daughter, Liz. They went through many trying times, but they showed infinite patience with me, showed enormous tolerance of the many hours I spent with this thesis, and continued to encourage me all along the way.

Patricia K. Lawlis

List of Figures

Figure		Page
1	Structure of the Math Unit	14
2	Floating Point Representation	26
3	Declaration Using Free Union Variants .	29

List of Tables

Table		Page
I	Description of MATHPAK's Representation Functions	35
II	Description of MATHPAK's Arithmetic Functions	36
III	Description of MATHPAK's Exp/Log Functions	37
IV	Description of MATHPAK's Trig Functions	38
V	Description of MATHPAK's Support Routines	39
VI	Description of IMPLMENT's Main Functions	42
VII	Description of IMPLMENT's Support Routines	44

Abstract

This project involved designing a unit of mathematical functions for an Ada language environment. The design includes the basic arithmetic operations and the transcendental functions. It uses modularity and defines both integer and floating point number representations. The main features of the design are its use of any specified precision (referred to as arbitrary precision), its implementation independence, and its structure as two separate Ada-like packages. Two packages are used to permit an implementation to increase the unit's efficiency, if desirable. The main package, MATHPAK, sets up all of the unit's functions and performs all but the lowest level arithmetic operations. The secondary package, IMPLMENT, performs the machine level arithmetic operations, and its implementation independent version can be replaced by a more efficient, machine dependent package without requiring changes in MATHPAK.

A partial implementation of the design was included as part of the project. This implementation was written in Pascal, since validated Ada environments did not yet exist. It is independent of any particular machine, and it implements arbitrary precision addition and subtraction for both integers and reals.

I. Introduction

The new computer language, Ada, promises to be the most significant development in the area of computer languages since the emergence of the first high level language, Fortran, in the 1950s. With it comes a new challenge. Good software tools are needed to aid in the establishment of software production facilities which can generate Ada software.

In particular, a unit of mathematical (math) routines is essential to every language environment. Traditionally, such math units have used the precision of the computer hardware to perform calculations. In other words, if the hardware only supported 60 bit floating point calculations, then all floating point calculations used 60 bits. If both single and double precision were available, say 32 and 64 bits, respectively, then users only got to choose between 32 and 64 bit precision. On the other hand, Ada is designed to encourage flexibility, and it allows for "tunable" precision. But the ultimate in precision flexibility is to allow the user to define any desired precision. This paper calls this concept "arbitrary precision", and it discusses a project which designed an arbitrary precision math unit for Ada.

Background

Around the middle of the last decade, the Department of Defense (DoD) realized that it was faced with serious problems with its extensive computer software. Maintaining the software had become extremely expensive, but even at $3.5 billion per year, much of the software was unreliable. Most of the problems centered around the fact that the DoD used more than 450 languages in its wide variety of applications. Particularly in the many weapons systems, where small computers are embedded into the systems as integral functional parts, software maintenance and reliability were a nightmare. Different systems used different computers and, consequently, different languages. Thus, general-purpose software tools typically were not developed for embedded systems, and the normally tight time schedules for development of these systems resulted in few tools of any kind ever being developed for any of them. Many systems were lucky to have working compilers in time to test the applications software as it was developed. The result was many poor quality systems with maintenance costs running higher than system development costs (Refs 6:7, 18:25-26, and 29:74).

To solve these problems, the DoD decided that it needed a common high-order computer language that could be used for all embedded software. It then set out with a refreshingly new approach to getting such a language. After determining the many requirements for the language

and establishing that no language meeting those requirements was then in existence, the DoD decided to design one of its own. But it did not just try to get a contractor to develop the language with a low budget and tight time constraints. Rather, the DoD decided to maintain maximum control and allow as much time as would be required to do the job right. It took the necessary time to carefully design a language which would meet all of its varied requirements for embedded software. And it went further. It solicited ideas, criticisms, and comments from computer professionals worldwide, and it revised the language specifications again and again based on the feedback received (Refs 6:8 and 38:27).

The language requirements were finalized in the "STEELMAN" report in June of 1978 (Ref 15). Then in July of 1980 the proposed standard reference manual was released (Ref 13), followed by a formal definition of the language in November of the same year (Ref 22). As for the language described in these documents, Brender and Nassi cover it succinctly:

> "The language draws on many years of research into algorithmic languages and programming methodology. It incorporates and directly supports modern programming concepts of abstraction and modularization, separate compilation of program units without loss of program-wide checking, concurrency, and features for efficient systems programming" (Ref 5:16).

This language acquired its name, Ada, late in its development. Sources indicate that Ada was named after

Ada Augusta Byron, Countess of Lovelace, who was associated with Charles Babbage and was the world's first programmer (Refs 37 and 38:27). The name, Ada, somehow perfectly represents the simple, yet important, purpose for the existence of this language: Ada will replace all other computer languages to become the only language used for software in DoD embedded systems.

Because Ada was designed so systematically and because the DoD is counting on it to become the sole provider for so many critical systems, nothing could be left to chance. Along with the language design came another set of DoD requirements for support environments for Ada programming. These also went through much world-wide scrutiny and many revisions before the final "STONEMAN" document was released in February of 1980 (Ref 14). Then in October of 1980, a compiler validation guide was published (Ref 33) to aid language implementors to insure that their Ada compilers are correct according to the language definition. This guide represents an extensive effort which helps to promote standardization among language implementations (Ref 20).

With so much careful work on a language and with the existence of so many documents defining the language and its environments, it is not surprising that the numerics of Ada are also carefully specified and constrained (Refs 13:3-8 - 3-18, 15:7 - 8, 22:3-27 - 3-36, 24:5-1 - 5-17, 33:3-16 - 3-29, and 37). These numeric specifications,

and how they relate to arbitrary precision, are the major concerns of this thesis.

The Problem

Many projects are presently underway which are working on the various aspects of the Ada software environment. However, little has yet been said about producing software either to implement the numeric requirements of the language or to provide necessary math functions for use by Ada programmers.

Most computer programs use math operations or functions. For some, simple arithmetic operations such as addition and multiplication are all that is needed, while for others exponential or trigonometric (trig) functions or the like are required. Typically, for different computers and different languages, these operations and functions are provided in different ways. Furthermore, the actual functions provided, as well as the levels of precision provided for each, usually vary greatly from one implementation to another.

One of the goals of the Ada design is to provide software portability (Refs 15:4, 18:30, 35:19, and 38:28). Portable software can be run without modification on any number of different computers and will produce essentially the same results on any one of them. This concept calls for the development of implementation independent software.

Scope

In keeping with the Ada goals, the objective of this thesis project is to design an implementation independent unit of floating point math functions which could be used in any Ada environment. The unit would define number representations and provide all necessary arithmetic operations, beginning with basic addition, subtraction, multiplication, and division. Additionally, it would provide all the standard math functions needed by any implementation, such as changing number representations and computing transcendental function values. It would work on a machine using any internal precision scheme and would use any desired precision in its calculations. It would also be expandable so new functions could be added as found necessary.

Of course, the ultimate goal of this project is to implement this design in an Ada environment. However, such a goal is far too ambitious for one masters thesis project. Hence, the project described in this paper is the initial effort of designing the unit of math functions and implementing the basic arbitrary precision capability of the design to prove its feasibility. It involves:

1. Designing the basic structure of the unit,
2. Researching algorithms for use in performing the various operations and functions,
3. Determining and laying out the number representations to be used, and

4. Implementing and testing the basic arbitrary precision capability.

Completing the implementation of all the math functions will be left for a follow-on thesis project.

Assumptions

In laying out this project, the assumptions have been kept to a bare minimum. It is assumed only that the machine used to execute the math unit:

1. Uses at least eight bits (one byte) in its internal representation of an integer number,
2. Performs the basic integer operations of addition, subtraction, multiplication, division, and modulus on one word integer numbers,
3. Performs the basic relational comparisons for equal, not equal, less than, and greater than for boolean words, and
4. Performs the basic boolean operations of AND, OR, and NOT for boolean words.

General Approach

The Air Force Institute of Technology (AFIT) presently has a preliminary Ada compiler, written in Pascal, which was designed by Capt Alan (Ray) Garlington as a thesis project (Ref 19). Since the completion of that thesis only a year ago, other AFIT thesis students have been working to expand its capabilities and to produce other software which could complement it in an Ada

language environment. The math unit is intended to eventually become a part of that environment as well.

It would be most preferable tc write the unit in Ada, since eventually all Ada software tools should be in Ada. However, since Ada is so new, complete and validated Ada compilers do not yet exist. In order to be able to test the software, and to maintain consistency with the developing AFIT Ada environment, this first version of the math unit will be written in Pascal. But the design will be specifically aimed at Ada constructs, and the Pascal implementations will be as Ada-like as possible. This will not be too cumbersome since Ada is modeled after Pascal.

Sequence of Presentation

To begin describing the development of this project, Chapter II looks at the project design and the modular structure of the design from top to bottom. It also discusses the many algorithms available in the literature for performing the math functions of interest and considers the issue of number representations for supporting arbitrary precision.

Chapter III is an in-depth look at the portion of the design which has been implemented. In developing this software, time constraints limited the amount which could be completed. However, the basic structure was completed, along with the modules which determine the representations

of the numbers and set up the arbitrary precision operations on these numbers. This chapter discusses these modules and the methods used to make the unit independent of any particular implementation. It also describes the functions laid out in the structure and takes a look at testing the software.

Finally, Chapter IV discusses conclusions and recommendations concerning this project. Since the overall objective was so ambitious, much hope remains for the future. The chapter considers the completion of the unit in Pascal as well as its translation into Ada. It also looks at how the unit may be used, improved, and eventually enhanced.

II. Project Design

The design of this project reflects the modularity of a top-down structured approach. This chapter looks at the design, starting with the top level and working all the way down to the bottom levels. It also considers the various algorithms available for performing the calculations to be implemented by this design and the number representations used to implement the arbitrary precision of the math unit.

The Structure

With the top-down design approach, the first step was to determine the highest level perspective of the math unit, and design its interface with the "outside world". From there the design worked its way down through the intermediate levels and finished by determining the functioning of the bit manipulations which occur at the very lowest level.

Top Level. Considering the entire math unit as a module performing the abstract function of transforming numbers, the first decision was to determine how a user would interface with this module. The kind of inputs the module will need, and what it should output had to be determined.

Since a language environment is closely tied to the machine used, languages model the machine. Both Pascal and Ada are stack-oriented languages which typically run

on stack machines, and thus they both model the machine memory as a stack. The ideal situation is for the support routines of the environment to communicate directly with the stack. Eventually, it may be possible for the math unit to use the physical stack mechanism in an Ada environment implemented on a stack machine. For now, this math unit design models the machine in software as an array of integers. This array is viewed as the stack, and it is a useful construct for handling arguments of dynamically changing sizes, such as the number arguments of arbitrary precision. The stack provides the means for passing arguments to and from the math unit.

In addition to the arguments themselves, other information needs to be communicated between the user and the math unit. Specifically, the unit needs to know where it can find the arguments on the stack and what it is expected to do with them. Also, with arbitrary precision, it needs to know the size of each argument. The user then needs to know where to find the result on the stack and if any errors occurred while performing the calculations.

The most convenient way to model the communication of precision information is for the using program to place a number, representing the precision, on the stack with the argument. The unit will receive both a stack pointer and a function code as calling parameters and pass back both a stack pointer and an error code. In Pascal, the stack pointer will be an array index into the memory stack, and

the function code will be a number indicating which specific function of the many provided by the math unit is to be performed. The error code will be a number corresponding to a particular predefined error message. The math unit will not need to send back the precision of the answer since that can be determined by the caller from other information.

One other final matter at this top-most level of design was to pick a name for this math unit. Since it will correspond to an Ada "package" construct, it was called MATHPAK. Thus, MATHPAK was to become an implementation independent "package" which would transform numbers in a manner which still had to be determined.

Intermediate Levels. Once MATHPAK's interface with the outside world was designed, the next step was to move to the next lower level. This was the design of the overall structure of the MATHPAK itself.

Still using the top-down approach, the MATHPAK was designed in such a way that the lower level constructs would be logical extensions of the higher level constructs. In addition, the modules in the MATHPAK were organized into two general groups: (1) main functions and (2) support routines.

The main functions are those specific MATHPAK functions which can be requested by the user via the function code. The function code determines which one of these functions will be executed, and each of the

functions calls upon specific support routines to help it perform its function. The functions are also organized into groups, each group representing a particular type of function. The groups are:

1. Representation,
2. Arithmetic,
3. Exp/Log, and
4. Trig.

See Figure 1 for a graphic representation of the MATHPAK within the language environment.

The design attempts to use only small modules, each of which performs only one task. Many of the ideas for designing such modular software tools were taken from Kernighan and Plauger (Ref 26). The modules of the MATHPAK support routines are its tools. Painstaking effort has been put into trying to make each of these modules perform its task in such a general way as to permit its use in more than just one specific application.

Looking at the overall MATHPAK module, it was designed to take in the function code calling parameter, determine which of its main functions corresponds to this code, execute that particular function, and return the result of that execution. MATHPAK defines both the number representations to be used in the language environment and the manner in which arithmetic is performed using them. Hence, not only does MATHPAK need to be able to do arithmetic when needed by a trig function, for example,

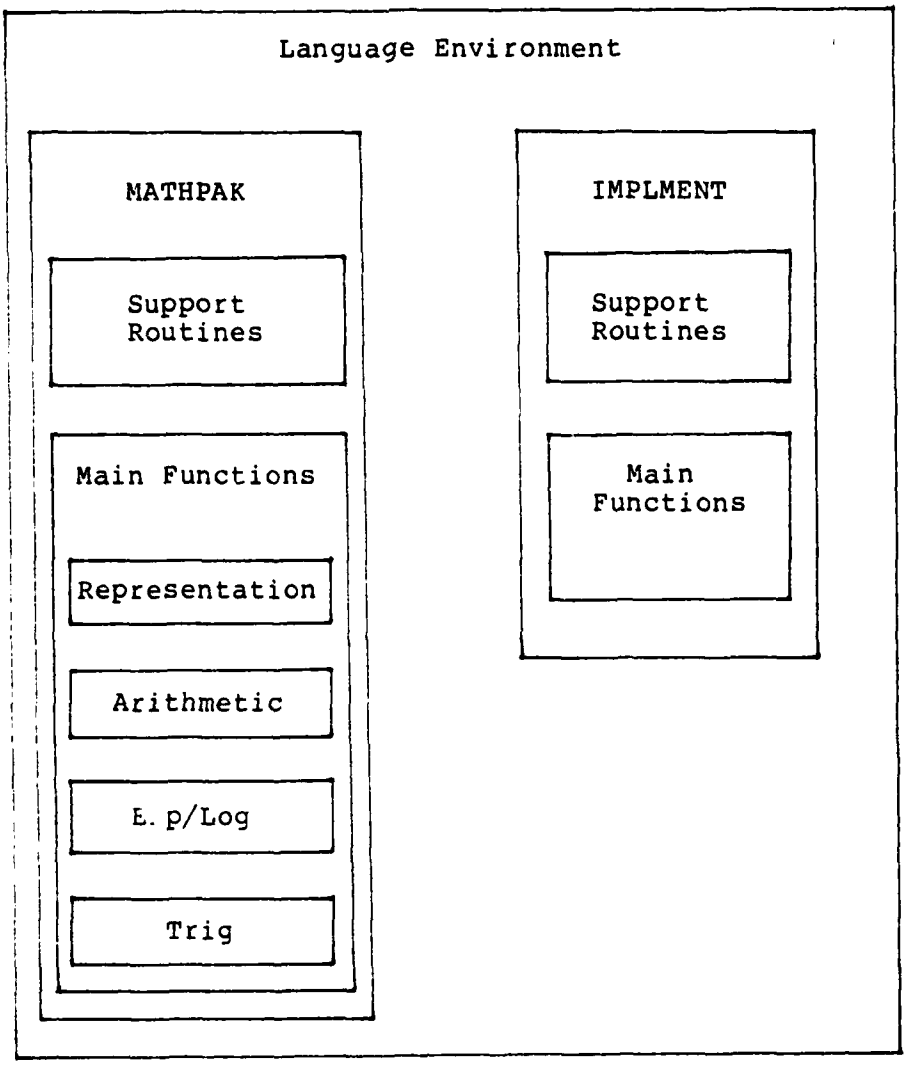


Figure 1. Structure of the Math Unit

but it also needs to be able to do ALL the arithmetic for every program in the language environment.

Progressing through successive intermediate levels of the design, each module is viewed as performing one specific task at that level. When it needs to use lower level tasks to accomplish this one task, it calls upon lower level modules to perform the lower level tasks. By far the most difficult problem throughout this effort was to maintain generality in the design of each module.

Lower Levels. At the lowest levels of the design, where the actual number transformations occur, the complexities involved with number representations had to be dealt with. When looking at bit manipulations, particular machine characteristics are very important. The actual implementation of the most basic operations, such as division, varies from one machine to another. To design the MATHPAK with only one particular implementation of such operations in mind is to design in unnecessary inefficiencies for different machine implementations. If, for example, a particular machine has hardware designed specifically to perform division with a high degree of efficiency, there is no good reason to impose artificial constraints on this machine. On the other hand, since other machines don't have such fancy hardware, such an implementation can't be required.

This dilemma was solved without sacrificing the goal of machine independence for the MATHPAK. A separate "package", called IMPLMENT, was designed to perform the operations which could be tailored to a particular

implementation, if desired. This package, like MATHPAK, will reside in the language environment, but only MATHPAK needs to know of IMPLMENT's existence. The design of IMPLMENT will be valid for any machine, but the actual code could be changed for any particular implementation without requiring any changes in MATHPAK.

IMPLMENT was designed with the same top-down approach as MATHPAK. First, it was determined that IMPLMENT will interface with MATHPAK in much the same way that MATHPAK interfaces with the outside world. Number arguments and the precision of the numbers will be passed back and forth on the stack. IMPLMENT will receive a stack pointer and function code as calling parameters. It will also send back a new stack pointer and an error indicator as parameters of the call.

IMPLMENT also has the same general structure as MATHPAK. At the top level its purpose is to determine which of its main functions is to be performed, according to the function code received as a calling parameter, and then to execute that function and return a result from that execution. It is laid out in small modules, each of which does one particular job. And the modules are also organized into the same two groups as MATHPAK: (1) main functions and (2) support routines. See Figure 1 for a graphic representation of IMPLMENT within the language environment.

The general structure of computer hardware determined the particular functions IMPLMENT would perform. Although each different type of computer is unique in certain ways, similarities also exist. The basic instruction sets, which reflect the hardware design, all perform certain basic operations, although in different ways and to different extents. As a basis for the choice of functions for IMPLMENT, Osborne's extensive summary of instruction sets for various microprocessors (Ref 31) was consulted, as was this writer's experience with instruction sets of several large mainframe computers. Patterns are obvious, even across such a diverse spectrum.

At the machine instruction level, all computers considered perform much the same basic arithmetic and logic operations. The logic operations can usually be simply and uniformly invoked from high level language constructs. However, the actual bit manipulations involved in performing arithmetic generally involves complex coding in high level language instructions, and this typically introduces gross inefficiencies. These basic arithmetic operations, then, are the functions which IMPLMENT was designed to perform. Thus, each implementation can address the subject of the efficiency of performing these operations on its particular machine, and each can produce its own version of IMPLMENT, if desired.

Of course, as more work is done on the transcendental functions in MATHPAK, additional implementation dependent requirements could surface. But IMPLMENT's modular design can easily handle additional functions any time new developments require such additions.

Algorithms

All of the functions in both MATHPAK and IMPLMENT are implemented by the execution of algorithms. Much of the research for this thesis involved finding algorithms for all of these various functions. At the start of the research, the implementation of the algorithms was expected to be a part of this thesis effort. But as the thesis definition changed, it was evident that there would be little time to actually use most of the algorithms. However, the algorithm research stands as an important first step toward the eventual implementation of the MATHPAK functions. Hence, outlined below are the sources found in the research, as well as any conclusions which may have been drawn about the usefulness of the algorithms discussed.

A good starting place was the CRC Standard Math Tables (Ref 12). This book contains complete algebraic formulae for all the transcendental functions, and it also has extensive tables of values for these functions. This book is not recommended for developing the computer algorithms, since it does not in any way address computer

calculations. However, it is a good source for determining actual values when testing the functions.

To find computer algorithms, a good source is the Collected ACU Algorithms (Ref 3) which contains a rather complete list of algorithms for the transcendental functions. Some of the algorithms are contained in the book itself, while others are found in other sources, which are referenced (Refs 8, 30, and 39). All of the algorithms of interest to this discussion are written in Algol. Since it is a block structured language, much like both Pascal and Ada, these algorithms could probably be translated into either language without much difficulty. However, other sources may well be even better.

The Software Manual for the Elementary Functions (Ref 10) is intended as a complete "cookbook" for implementing transcendental functions on a computer, and it appears to have a better approach than the Algol procedures discussed above. The mathematical basis for its algorithms can be found in a book called Computer Approximations (Ref 21). From the first source, one can find complete descriptions of how to implement algorithms for most of the transcendental functions. From the second source comes the theory, which may be needed for extending the algorithms to arbitrary precision.

One other source was investigated, and it led to an apparent dead end. In Benchmark Papers in Electrical Engineering and Computer Science is a section on computer

arithmetic which contains algorithms for calculating transcendental functions (Ref 4). However, the algorithms discussed here, called CORDIC algorithms, were derived for special applications. They do not appear to be useful for MATHPAK applications.

None of the sources discussed above treats arbitrary precision. Typically, an algorithm is either for single or double precision. The "cookbook" goes a little farther and usually discusses four ranges of precision levels for each function. It concerns itself with precisions as large as 65 bits, depending on the function. However, purely arbitrary precision in transcendental function calculations has apparently not yet been addressed in the literature.

Although various methods are used in the above sources for calculating the transcendental functions, they are all similar from an implementation point of view. In order to execute these functions in arbitrary precision, the math unit would essentially need to be capable of executing arbitrary precision arithmetic operations (addition, subtraction, multiplication, and division) and other arbitrary precision functions which would be implemented using arbitrary precision arithmetic.

As for the basic arithmetic operations, few sources were found which treated this type of algorithm in depth. However, Knuth gives a rather thorough treatment of specific algorithms (Ref 28:180-192). Both Knuth and

Kahan also give good general discussions on algorithms and floating point numbers. They treat such important topics as accuracy in floating point arithmetic and the price that must be paid for it (Refs 25, 27, and 28).

Arbitrary precision integer arithmetic is, for the most part, a straightforward process of chaining machine-provided arithmetic operations on pairs of words. Raskin used this type of technique for arbitrary precision division (Ref 32), but the principle applies equally well for any arithmetic operation. Hence, once any one arithmetic operation is implemented for arbitrary precision integers, the others would follow using similar techniques.

To properly treat floating point arithmetic, the key is to implement the addition algorithms first. This holds for arbitrary precision arithmetic as well as for fixed precision, although the arbitrary precision implementation is somewhat more involved. Floating point addition involves separating the exponent and fractional parts of the arguments and then doing numerous shifting operations, both for scaling numbers to align the radix points, before adding the fractional parts, and to normalize the result afterward. The value of the exponent must also be changed accordingly when shifting, fractional parts must sometimes be complemented, and numerous checks for overflow and underflow must also be included (Ref 28:182-187).

Once the addition capabilities are implemented, subtraction follows immediately and multiplication and division are entirely straightforward. For subtraction, the sign of the second argument is changed, and then addition is performed. For multiplication (division), the exponent and fractional parts are separated, the exponents are added (subtracted), the fractional parts are multiplied (divided), and the result is normalized. Multiplication and division of fractional parts use machine operations, while addition and subtraction of exponents use the addition function. Separating the exponent and fractional parts and normalizing results also use capabilities already implemented for addition (Ref 28:185-190). The additional capabilities required to use arguments of arbitrary precision are the abilities to deal with arbitrary precision representations and the repeated execution of an operation. Clearly, these capabilities would be required for an implementation of addition, and they would thus be available for other operations once addition is implementetd.

Therefore, once the arbitrary precision integer and floating point addition algorithms have been implemented, the math unit's arbitrary precision design will be proven valid and feasible. Then the implementation of the remaining arithmetic operations will be straightforward using the capabilities already available for addition. Similarly, the transcendental functions can be implemented

in a straightforward manner once the arbitrary precision arithmetic operations are fully functional.

Number Representations

When looking at algorithms for the basic arithmetic operations, the representations of the numbers involved are of utmost importance. These representations must also be known when designing the number representation routines. So one of the design decisions for MATHPAK was what representations would be used, both for integer and floating point numbers.

Since representations differ for different computers, no absolute standards exist to go by. And clearly, each computer must not use its own representations because MATHPAK and IMPLMENT must be able to manipulate the bit patterns of numbers while performing their functions. But Ada provides specific numeric requirements, so these have been used as a basis in determining the representations to be used in the design.

Ada requires that a floating point number have a fractional part between 0.5 and 1.0. In other words, the radix of the number must be 2. Another requirement is that the exponent be an integer in the range from -4 to +4 times the number of bits in the fractional part. Floating point types can be declared with or without a range and a specific number of decimal digits D. If D is specified, it corresponds to a specific minimum number of binary

digits (bits) B, where B is the integer next above D*ln(10)/ln(2). D can also be made to correspond to a predefined floating point type, FLOAT, LONG FLOAT, or SHORT FLOAT. These types are just predefined precisions of an implementation, and the specification of D in a type declaration would just guarantee a minimal precision when the specified type is made to correspond to a predefined type (Refs 13:3-14, 15:8, 22:3-31, 24:5-6, and 33:3-20).

The MATHPAK concept of arbitrary precision is consistent with these Ada requirements, but it extends the precision possibilities somewhat so that a specification of D can be made to correspond to one of numerous types rather than being limited to just three. MATHPAK's only limit on the precision of a number is the actual size of memory in the machine being used, and MATHPAK works with precision in one word increments. In other words, if a user specified the precision of a number to be longer than three words but shorter than four, MATHPAK will use four full words to represent the number.

A series of Computer magazine articles discussing the new floating point standard proposed by the Institute of Electrical and Electronics Engineers (IEEE) was also consulted when determining the number representations for MATHPAK to use (Refs 1, 9, 11, and 23). This proposed standard is compatible with the Ada requirements, but it also goes into detail about the floating point number representations. Since this proposal is compatible with

Ada and since it is the closest thing to a floating point standard, it was used as a basis for the floating point representations for MATHPAK. Of course, since the standard does not apply to arbitrary precision, it had to be modified some. But Ada's restrictions on the lengths of the exponent and fractional parts of a floating point number made that modification rather clear-cut.

The floating point representation used in MATHPAK is pictured in Figure 2. The first bit, S, is the sign of the number, where 0 means positive and 1 negative. E is the exponent, and its length is determined by the Ada requirement that the value of the exponent must range from -4 to +4 times the number of bits in the fractional part, F. The exponent is biased by 2 raised to a power which is one less than the number of bits in the exponent. The effect of this bias is that the value of the exponent will never be negative, and thus no sign bit is required for it. Of course, the lengths of both E and F can vary as the length of the floating point number varies. Therefore, this representation is valid for floating point numbers of any precision.

As for integer representations, Ada defines no specific requirements. The two's complement form was chosen for integers because of its wide usage and because it supports simplicity in calculations. In the two's complement form, a positive integer is represented by a zero sign bit followed by the bit pattern for a binary

Figure 2. Floating Point Representation (Ref 11:71)

number whose value is equivalent to the value of the given integer. A negative integer representation is found by taking the complement of the absolute value of the integer and adding one to it. Of course, this results in a sign value of 1, which properly indicates that the number is negative.

Now that the design of the MATHPAK, the algorithms considered for use in the design, and the number representations used in conjunction with the algorithms have been considered, it is time to look at the computer code implementing the basic arbitrary precision part of the design. This will be done in the next chapter.

III. Software Implementation

This chapter discusses the MATHPAK and IMPLMENT software which was initiated as a part of this thesis project. The software was developed on an Apple II Plus computer using Apple Pascal, which is a dialect of UCSD Pascal (so named because it originated at the University of California at San Diego). The Apple II is an 8 bit machine, but it performs 16 bit integer operations. Thus, for the math unit, an Apple word is 16 bits.

It is not common to work at the bit level in a high level language, and implementation independence is typically an elusive goal for software. Hence, the process of manipulating bits and the issues of implementation independence will be considered first in this chapter. Then the status of both the MATHPAK and IMPLMENT packages as of the end of the work on this thesis will be discussed. Appendix A shows the structure of the presently implemented modules of MATHPAK, and Appendix B shows the present structure of IMPLMENT. These structures form the framework on which future work on the math unit can be built. The chapter will end with a discussion of testing. It will look at both what testing has been done and what will need to be done to prove that the math unit works properly when the project is completed.

Bit Manipulations and Implementation Independence

Although Pascal is a rather strongly-typed language which has no specific constructs for bit manipulations, the language does provide ways for a programmer to "tinker" at the bit level. Pascal records may contain variant parts, parts that are interpreted differently in different situations. These variant parts can contain tag fields to identify how they are to be interpreted. In this case, the contents of a particular record will always be interpreted as compatible with the tag fields. However, variant parts which do not contain tag fields, called free union variants, are also legal in Pascal. In this case, the same data in the same record can be interpreted as being of different types in different statements of the same program (Refs 7:454-455 and 36:286-287).

Of course, free union variants can be dangerous because they short-circuit the built-in Pascal type-checking safeguards. But they provided a useful tool for this project because they permit the math unit to do bit manipulations.

Figure 3 is an example of a type declaration which uses free union variants. This is a portion of the IMPLMENT code which declares ARITHREG as type REGISTER. ARITHREG is one machine word in length. When performing integer operations, ARITHREG can be interpreted as an integer. However, at other times it can also be

interpreted as a boolean, an array of bytes, or an array of bits, depending upon how it is referenced. Thus, four representations are possible for the same contents of ARITHREG.

```
CONST
    LSBITNUM = 0;
    MSBITNUM = 15;
    BYTESIZE = 255;

TYPE
    REPS = (E,F,G,H);
    BITE = 0..BYTESIZE;
    BYTENUMBER = 0..1;
    BITNUMBER = LSBITNUM..MSBITNUM;
    BYTEARAY = PACKED ARRAY[BYTENUMBER] OF BITE;
    BITARAY = PACKED ARRAY[BITNUMBER] OF BOOLEAN;
    REGISTER = RECORD CASE REPS OF
        E: (INT: INTEGER);
        F: (BOOL: BOOLEAN);
        G: (BYTE: BYTEARAY);
        H: (BIT: BITARAY);
        END;

VAR
    ARITHREG: REGISTER;
```

Figure 3. Declaration Using Free Union Variants

In the sample code, the four identifiers in REPS are just dummy declarations needed to show that four variant types will be declared. The identifiers which are used in the program to select the desired representations are the descriptive terms INT, BOOL, BYTE, and BIT. Hence, ARITHREG.INT refers to an integer. Similarly, ARITHREG.BIT[MSBITNUM] refers only to the most significant bit of ARITHREG and it is operated on as a boolean value.

The numbers declared as constants in Figure 3 are specific for the Apple II. These specific numbers are in the constant declarations so they may be changed for different implementations without requiring other changes to the IMPLMENT package. For this Apple implementation, a word has 16 bits, numbered 15 to 0 from most to least significant. The two half-words (BITEs) are numbered 1 and 0 for most and least significant, respectively. BITE is declared as an integer in the range from 0 to the largest number which can be held in half a word, which, in this case, is 255. Since this value needs exactly one half word, a packed array of BITEs uses exactly one word (as does an integer, a boolean, and a packed array of 16 bits).

Although care has been taken to make the Pascal version of both MATHPAK and IMPLMENT implementation independent, reservations must be stated in this respect. First of all, each package requires that the constants be set for the implementation, so a change is required for each implementation, but it can be done easily. Even after setting the constants, however, with the rather tricky bit manipulations performed by portions of the code, complete machine independence still cannot be guaranteed.

Most computer languages have numerous dialects, and Pascal is no exception. Throughout MATHPAK, either the Pascal dialect which meets the proposed standard of the

International Standards Organization (ISO) has been used, or else the documentation for the non-standard module states what work is required to make the final version meet the ISO standard. Tiberghien's The Pascal Handbook (Ref 36) is the authority which has been used to determine the correct usage of ISO Standard Pascal. But to the extent that the free union variant constructs of ISO Standard Pascal can't be executed in the environment of other dialects, the chances of MATHPAK and IMPLMENT working without modification diminish.

Another possible problem is that one characteristic of booleans which has been taken advantage of in the Pascal code of this project could possibly work differently on other machines than on the Apple. Boolean operators are usually assumed to perform single TRUE and FALSE operations on only the least significant bit in a word. However, on the Apple, boolean operations are performed bitwise on all 16 bits of the operands, and this fact has been used in some of the boolean operations performed in the packages. Therefore, these packages would not execute correctly should they be run on a machine which works differently with booleans.

Before leaving the subject of bit manipulations, some comments need to be made about doing these same bit manipulations in Ada. Since one of Ada's design criteria was for it to be able to handle systems software, Ada has specifically designed ways to handle bit manipulations.

These are much less awkward and far more obvious than the methods which have been used in Pascal. Hence, when the Pascal code is translated to Ada, one will not have to resort to unsafe "tricks" as in Pascal (Refs 13:1-1, 17:18-23, 22:1-1, and 34:29,80).

The Ada designers have also gone to great lengths to make sure the language is completely standardized and independent of any implementation (Refs 15:21 and 17:49-51). Hence, to the extent that this standardization is achieved, the math unit in Ada should have no problem with different language dialects causing deterioration of the goal of implementation independence.

MATHPAK

This section, will look at the presently implemented form of MATHPAK in some detail. The reader can refer to Appendix A for an overall picture of the structure of the MATHPAK modules and to Appendix C for a complete listing of the MATHPAK to its present point of completion.

In the declarations of the MATHPAK listed in Appendix C, the constants are set to the correct values for an Apple II implementation. These constants provide the means for setting up the MATHPAK for any particular implementation, and they permit the remainder of the MATHPAK code to be independent of the implementation. With this set-up, any time MATHPAK is implemented in a new environment, these constants must be set to correspond to

that environment. They tell MATHPAK the machine wordsize, how the bits are numbered in a word, how the global memory stack will be built in this environment (high to low memory or vice versa), and the maximum size allowable for a number in this environment. Once this is done, MATHPAK can be compiled, and it's ready to go.

MATHPAK declares some free union variant records, as described in the previous section. One of these is an array of three numbers which MATHPAK uses as a work area to perform bit manipulations on arguments. Each of these is the maximum allowable size for a number in this environment. Note, however, that such a large work area would not be necessary if MATHPAK could do bit manipulations on the stack. In a later Ada version, if MATHPAK can communicate directly with the machine memory stack, this separate large work area would not be necessary.

MATHPAK also uses a free union variant record to set up an array of masks. These are used for masking to ensure that only the desired bits of a word are changed when MATHPAK writes bit values to words in the work area. The masking is not necessary in the Apple UCSD Pascal used for the MATHPAK development, but it is necessary for a Standard Pascal implementation.

MATHPAK's main functions are those which can be requested by the calling program via the function code. The transcendental functions are not yet operational, but

most of the number representation functions are complete and the functions which perform arithmetic operations for arbitrary precision are fully operational (see Appendix A). The most important arithmetic operation to develop for this math unit was addition. It is the most complex of the operations to implement, and it establishes the basis for the other operations because the others perform steps which must be developed for addition (Ref 28:187). Arbitrary precision addition is completely operational for both integers and floating point numbers, and, with this, the design of the math unit has been proven valid and feasible. See the section on Algorithms in Chapter II for a more detailed discussion of this point. With the completion of addition, subtraction was also easily completed as the addition of the inverse of the second argument. Time did not permit the completion of multiplication and division as a part of this effort. Although their implementation is straightforward now that addition is operational, they each require the integration of several steps, and this is considerably more time-consuming than the one-step effort required to complete subtraction using addition.

All of the MATHPAK functions are described in Tables I through IV. Each of the tables lists the functions for one of MATHPAK's groups of functions, and in each of these tables, INT refers to an integer type and REL to a floating point (real) type. Also, for groups of functions

with names that differ only by I and/or R at the end of the name, I refers to an integer argument and R to a real argument. If a name has two of these letters at the end, the function takes two arguments in the order indicated by the letters. Where a function takes both an integer and a real argument, the result is always real.

Table I describes the names and tasks of the Representation Functions. Each of these can perform a change in the representation of a numerical argument of the correct type.

Table I

Description of MATHPAK's Representation Functions

Name	Function
FLOAT RELTOINT	Change a number from one representation to another
INTSIZECHANGE RELSIZECHANGE	Change the precision of a number
ABSVALI ABSVALR	Replace a number by its absolute value

In Table II, the Arithmetic Functions, which perform all of the basic arithmetic operations usually associated with the +, -, *, and / symbols, are named and defined. All of these functions yield a result of the same type as their input arguments, unless integers and reals are mixed, and then, as stated above, the result is real.

Table II

Description of MATHPAK's Arithmetic Functions

Name	Function
PLUSII PLUSRR PLUSIR PLUSRI	Add two arguments
PLUSI PLUSR	The identity operation which returns the argument unchanged
MINUSII MINUSRR MINUSIR MINUSRI	Subtract the second argument from the first
MINUSI MINUSR	Negate the argument
MULTII MULTRR MULTIR MULTRI	Multiply two arguments
DIVII DIVRR DIVIR DIVRI	Divide the first argument by the second
REMI	Divides the first integer by the second and returns the remainder
MODI	Finds the modulus of the first integer by the second

Table III shows the Exp/Log Functions, which include all functions associated with exponentiation and logarithms. The power functions, each using two arguments, yield a real result if types are mixed, and otherwise the result is of the same type as the input

arguments. The remaining functions in this group all take one argument and all yield a real result.

Table III

Description of MATHPAK's Exp/Log Functions

Name	Function
POWERII POWERRR POWERIR POWERRI	Performs exponentiation using the first argument as the base and the second as the exponent
SQRTI SQRTR	Finds the square root of the argument
EXPI EXPR	Performs exponentiation using e as a base and the argument as the exponent
LOGI LOGR	Finds the base 10 logarithm of the argument
LNI LNR	Finds the base e logarithm (natural log) of the argument

The Trig Functions are defined in Table IV, and each of these yields a real result. Functions using the trig functions cotangent, secant, or cosecant have not been included because these can be easily calculated from the tangent, cosine, and sine functions, respectively.

The support routines are the modules which perform the lower level tasks called for by the main functions. Many general support routines have been implemented, and most of these will also be used to make the remaining main functions operational. These general

Table IV

Description of MATHPAK's Trig Functions

Name	Function
SINI SINR	Finds the sine of the argument
COSI COSR	Finds the cosine of the argument
TANI TANR	Finds the tangent of the argument
ARCSINI ARCSINR	Finds the number whose sine is the argument
ARCCOSI ARCCOSR	Finds the number whose cosine is the argument
ARCTANI ARCTANR	Finds the number whose tangent is the argument
SINHI SINHR	Finds the hyperbolic sine of the argument
COSHI COSHR	Finds the hyperbolic cosine of the argument
TANHI TANHR	Finds the hyperbolic tangent of the argument

routines, and the types of tasks performed by each, are described briefly in Table V.

IMPLMENT

The original conception of IMPLMENT was as a module that had to be machine dependent and would probably need to be written in machine language. IMPLMENT must know when a carry or an overflow occur, for example, and to get

Table V

Description of MATHPAK's Support Routines

Name	Task
INITMASKS	Initialize bit masks
POPARG PUSHARG	Transfer arguments to and from the stack
WRITEVAL WRITEBIT BITVALFILL BITSTOBITS COPYEXPO	Transfer bit values
GETSTATZERO	Check for a zero argument
GETNEWSIZE SETWORDS EXPOSIZE CHECKEXPO	Calculate and check sizes of arguments and exponents
CHKHIWORDEND CHKLOWORDEND	Adjust word and bit index values at word boundaries
SHIFTRIGHT SHIFTLEFT	Shift a given number of bits over a given number of bits
INTBITREP RELBITREP	Prepares arguments in work area
COMPLMENT ABSVALINT	Negate number representation
SAMEINTPRECISION SAMERELPRECISION INTDECREASE RELDECREASE	Adjust the size of arguments and exponents
ADDARGS	Add arguments via IMPLMENT
FINDNON0BIT FIND1STFRACBIT	Determine bits to be used for representation changes
POWEROF CALCEXPO	Make exponent calculations

that type of information, it seemed necessary for IMPLMENT to get at the Apple's status register (Refs 2:12 and 16:120). The thought of doing any part of this project in machine language was not very attractive, however. IMPLMENT efficiency was not the main concern in this thesis effort. The main concern was the project goal, which was to design an implementation independent math unit and implement the basic arbitrary precision to prove the design's feasibility. In that light, a machine dependent IMPLMENT would be a necessary evil at best, for it would be needed to implement arbitrary precision, but it would not be part of a machine independent implementation.

After some thought, however, a better idea was conceived. If the arithmetic operations were done using only half a word at a time, IMPLMENT would be able to "see" carries, using the same Pascal bit "tricks" that were used in MATHPAK. And bits could be examined to determine overflows and underflows and the like. Hence, IMPLMENT could be done independent of the Apple, and it could be viewed as an integral part of the effort to produce an implementation independent math unit. Yet, at the same time, implementors would be able to replace this version of IMPLMENT, if desired, with one which could produce more efficiency for their particular situation.

The present form of IMPLMENT completes the implementation of arbitrary precision addition and subtraction in the math unit. Of the large number of support modules which have been implemented, most will also be used to support the multiplication and division functions, when they are completed. The reader can refer to Appendix B for an overall picture of the structure of the present modules in IMPLMENT and Appendix D for a complete listing of IMPLMENT to its present point of completion.

IMPLMENT is structured very much like MATHPAK. In its declarations in the listing in Appendix D, the constants are presently set to the correct values for an Apple II implementation, and once again, these constants can be changed to correspond to a new environment for a new implementation. This feature permits the remainder of IMPLMENT to be as independent of the implementation as MATHPAK.

Types have been declared using the free union variant records in much the same manner as in MATHPAK. The variables of type REGISTER act much like the registers of the machine (see Figure 3 for a complete declaration of type REGISTER). They are used to actually perform arithmetic operations. As with the MATHPAK, IMPLMENT also has declared large work areas, which would not be necessary if bit manipulations could be done on the stack.

Table VI

Description of IMPLMENT's Main Functions

Name	Function
INTADD RELADD	Adds two arguments
INTMULT RELMULT	Multiplies two arguments
INTDIV RELDIV	Divides the first argument by the second

IMPLMENT presently has six main functions. However, more may well be found necessary as MATHPAK's functions are completed. Of the present main functions, only two are completed (See Appendix B). These use algorithms and ideas from Knuth (Ref 28:180-187) to complete the MATHPAK addition functions and make arbitrary precision addition and subtraction operational for the math unit. All of IMPLMENT's main functions are listed in Table VI. Each of these functions takes two arguments of the same type (INT for integer and REL for real) and yields a result of that type. Note that subtraction is not included in IMPLMENT's functions because a subtraction operation in MATHPAK will always be turned into an addition operation before IMPLMENT is called. However, division is included because turning a division operation into multiplication is not usually desirable in computer arithmetic.

IMPLMENT has numerous support routines, and these perform many basic tasks. Again, structured much like MATHPAK, IMPLMENT already contains the key modules which will be used by the multiplication and division functions when they are completed. These modules, and the types of tasks each performs, are described briefly in Table VII. Note that these modules perform much the same types of basic tasks as the support routines in MATHPAK.

Testing

At this point in the development of the math unit, only preliminary testing has been done. However, as the development continues, much more testing will be required before the unit can be considered finished. This section outlines what type of testing has been done and what will need to be done in the future.

Only a few of the MATHPAK functions actually perform their complete functions at this time, and testing to this point has naturally centered around these. Through this testing, however, every presently completed module in both MATHPAK and IMPLMENT have been used to some extent. Just by doing "simple" addition and subtraction of various pairs of numbers, every module has been called and "exercised". This does not necessarily mean they all work perfectly, but it does mean they all work properly under many different sets of circumstances.

Table VII

Description of IMPLMENT's Support Routines

Name	Task
SIGNINTADD	Checks signs of arguments
ADDGENERAL	Adds two arguments
INTPREPADD RELFRACADDPREP RELEXPADDPREP	Prepares arguments in work area
CHKHIWRDEND CHKLOWRDEND	Adjust word and bit index values at word boundaries
RIGHTSHIFT LEFTSHIFT NORMLEFTSHIFT ADDNORMALIZE	Shift a given number of bits over a given number of bits
EXPFROMNUMBER ZEROOUTEXP EXPTONUMBER	Transfer bit values
EXPSIZE DECEXP CARRYONMSB	Calculate and check size of exponent
ARGTOWORKAREA WRKAREATOSTACK PUSHRESULT	Transfer arguments to and from the stack
COMPLEMENT	Negate number representation
CHKINTADD CHKRELADD	Check for errors in addition and make necessary adjustments to representation of result
MASKINIT	Initialize bit masks

A small driver program was used for the preliminary testing. It read an input file to get sets of numbers for setting up an environment and providing input arguments.

It then immediately wrote out all the input information for verification, and it also wrote out the bit patterns of all the input arguments. Next, it loaded the stack with one pair of input arguments at a time and called MATHPAK to add them, using the proper calling parameters for that particular test. After MATHPAK was finished and the driver got control again, it wrote out the bit pattern of the result as well as an error message, either indicating that no error had occurred or describing the type of error that was detected during the MATHPAK execution.

Additionally, write statements were placed at the beginning of each module to verify when each was called, and several write routines were also incorporated into both MATHPAK and IMPLMENT for testing purposes. With these, output was produced which "walked through" the execution, showing what modules were executed in what order and writing out the bit pattern of intermediate results at critical points along the way.

Several different combinations of numbers were used in the testing to verify that the completed functions are working correctly. The tests included adding and subtracting the following combinations of numbers:

1. Both numbers were of one word precision.
2. Both numbers were of the same large precision.
3. The numbers were of different precisions.
4. Both numbers were positive.

5. Both numbers were negative.
6. The numbers had opposite signs.
7. Both numbers were integers.
8. Both numbers were real.
9. One number was real and the other an integer.
10. The numbers were of significantly different magnitudes.

The tests also included instances where integers were changed to reals and carries were performed over half-word and word boundaries. Under all of the above conditions, the presently completed MATHPAK and IMPLMENT modules seem to be working properly.

As the development of the math unit continues, the type of testing done so far will need to continue. When the functions to be tested get above the level of simple arithmetic, it will become necessary to check the outputs against some authoritative source, such as the entries in the CRC Standard Mathematical Tables (Ref 12). However, additional sources will be required to verify calculations for numbers of large precision.

In addition to this preliminary type of testing which is done with a test driver program, the math unit must also be integrated into a language environment before testing is completed. It is expected that this will be done in an environment available at AFIT, and the math unit can then be interfaced with Garlington's compiler.

In Pascal, the math unit will actually become an integral part of the compiler because the compiler will have to know all of MATHPAK's function codes to be able to properly set up calls to the package. MATHPAK will also be used to perform all of the compiler's arithmetic. Additionally, the input and output routines of the environment will have to be able to deal with arbitrary precision numbers. They will probably also use some of MATHPAK's functions, particularly the representation functions, to assist in their tasks.

In Ada, where MATHPAK and IMPLMENT will actually be implemented as packages, the compiler will probably not need to know quite so much about MATHPAK. Users should be able to call for MATHPAK functions directly by using names set up in MATHPAK (probably generic names). With this new concept, and since Ada allows arithmetic operators to be defined as functions too (Refs 13:6-9 - 6-10, 22:6-20 - 6-21), MATHPAK may not need to be such an integral part of the compiler. A looser interface may be better. Actual experience with Ada environments will be needed to determine the best interface.

After complete testing with several different types of programs in the language environment, the math unit will be ready for conditional acceptance into the environment. Only an incurable optimist would not expect little bugs to be found now and then later on, but these could be handled as improvements for later versions.

Now that the reader has seen how the unit was conceived, designed, and partially implemented, it is time to discuss conclusions and recommendations. This will be done in the next, and final, chapter.

IV. Conclusions and Recommendations

Although the math unit is not yet completely implemented, some important conclusions can be drawn at this point. Numerous recommendations can also be made for the future of the math unit. This future includes the completion of the unit's modules, both in Pascal and in Ada, the use of the unit in language environments, and improvements and enhancements which can be made later on.

Completion

Before any real use can be made of this project, its present form must be completed. This means accomplishing several things:

1. Selecting algorithms for the incomplete functions. The algorithm discussion in Chapter II should be most helpful to this end.
2. Implementing the selected algorithms in Pascal. Many of the present support routines will be useful when doing this.
3. Completing the necessary revisions to ensure that the code is in ISO Standard Pascal. The modules that need revisions contain documentation describing what needs to be done.
4. Analyzing possible errors, and completing the unit's error processing capabilities. Much remains to be done in this area, particularly in MATHPAK.

5. Completely testing all main functions to ensure proper performance. This is an extension of the preliminary testing already done.
6. Integrating the unit into the environment of at least one operational system. Several of the systems available at AFIT would provide good proving grounds. The Avionics Laboratory at Wright-Patterson AFB, which presently sponsors Ada thesis efforts, makes a DEC 10 environment available to AFIT for such development work. A CDC 6600 is also available for AFIT use, and AFIT has its own VAX system which could also support this work. Of course, the entire AFIT Ada environment would have to be integrated into the system for this type of testing.

Completion also means more than just completing the present form of the math unit. It means translating the unit into Ada to complete the long-term goal of this project. This means accomplishing several additional items:

1. Modifying the design, as necessary, to take advantage of Ada constructs. These modifications should not affect the overall structure, but they will affect the specifics of how each objective in the design is accomplished.
2. Writing the Ada code. This can be thought of as a translation from Pascal, since the structure of

the code in each language should be basically the same. Changing MATHPAK and IMPLMENT from Pascal procedures to Ada packages will produce a need to make some changes to the method of interfacing them both with user programs and with each other. The error processing will also change significantly with the use of error handlers for exception processing in Ada. However, the overall function of the two packages and the information they exchange with user programs and with each other remain the same.

3. Completely testing all functions to ensure proper performance. Once again, this is the preliminary testing of the code.
4. Integrating the unit into at least one operational Ada language environment written in Ada. This would most likely be accomplished when the entire AFIT Ada environment is "translated" into Ada. However, when its Ada code is completed and fully tested, the math unit could be implemented in any Ada environment.

Use.

The math unit could eventually play an important role in the development of good software systems for the DoD. This project has created a design and enough of a Pascal implementation to prove the feasibility of the design.

The design could, when completed, prove useful in many Pascal environments. More than that, however, this math unit could provide a powerful tool in the Ada language environments provided for many future DoD software development efforts.

When all the functions called for in the present design are completed and proper testing has been done, the math unit could be implemented in a Pascal environment. As with integrating it into Garlington's Ada compiler, the unit would have to be integrated with the Pascal compiler, and this would require some work on proper interfacing. In particular, the compiler would have to be made aware of MATHPAK's existence, it would have to know MATHPAK's function codes, and it would have to declare a memory stack structure corresponding to the one MATHPAK expects to find. The system input and output routines would also have to be modified to read and write numbers of arbitrary precision to and from the necessary number representations used by MATHPAK.

This implementation could be done in a Pascal environment at AFIT. In addition to being a good place to test the math unit, AFIT is also a likely math unit user. The unit would provide an excellent tool for the engineering applications dealt with there, and the students would provide excellent feedback on how well the unit performs, as well as what improvements could be considered. Hence, a Pascal implementation would be worth

consideration, especially at AFIT.

It is also expected that once the AFIT Ada environment is integrated into one or more systems at AFIT, it will remain there to be used. This environment could eventually be expanded to a completely validated Ada operating environment, either entirely through research done at AFIT or through combining the software developed at AFIT with other software developed elsewhere for Ada language environments.

In the long run, the Ada version of the math unit could be implemented in other DoD Ada environments. If the unit's usefulness can be confirmed, then it could be put to use throughout the DoD, providing a powerful numerical capability for DoD software systems.

Improvements and Enhancements

Every popular piece of software goes through revisions based on the evaluations and needs voiced by its users. If this math unit becomes popular enough to warrant revisions, the most useful will probably be either to improve MATHPAK's efficiency or to break the unit up into smaller packages.

Working to improve the efficiency of a piece of software is a very common revision. Work in this area would surely be able to increase MATHPAK's execution speed. But spending a lot of time on such work for IMPLMENT is not recommended, since the implementation

independent version designed in this project is not intended to produce the best efficiency. An implementation really concerned with IMPLMENT's efficiency can write its own tailored version.

It is also possible that users could become concerned with the size of MATHPAK, particularly when many users will surely not need all of the functions included in MATHPAK. Future work could break up MATHPAK's various groups of functions (see Appendix A) into separate packages. In this way a user, or a software system, need only integrate into its library the packages needed for the application at hand.

If users begin to recognize the math unit as a valuable tool in a language environment, they will also surely ask for more. They will want MATHPAK to be able to provide every conceivable math function. This could include, but certainly would not be limited to, a complete spectrum of operations and functions for use on complex numbers and matrices. This would be a valuable enhancement for many engineering applications, for example.

The future for both Ada and the math unit is promising. This writer hopes to remain involved with the development of both, as software systems become more and more critical to the mission of the DoD.

Bibliography

1. "A Proposed Standard for Binary Floating-Point Arithmetic," Computer, 14: 51-62 (March 1981).

2. Apple II Monitors Peeled. Cupertino, California: Apple Computer Inc., 1981.

3. Association of Computing Machinery. Collected ACM Algorithms. New York: Association of Computing Machinery, 1976.

4. Benchmark Papers in Electrical Engineering and Computer Science/21: Computer Arithmetic, Part VI: Elementary Functions, edited by Earl E. Swartzlander, Jr. Stroudsburg, Pennsylvania: Dowden, Hutchinson & Ross, Inc., 1980.

5. Brender, R. F. and I. R. Nassi. "What Is Ada?" Computer, 14: 17-24 (June 1981).

6. Carlson, W. E. "Ada: A Promising Beginning," Computer, 14: 13-15 (June 1981).

7. Casseres, D. "Bits and Bytes in Pascal," Byte, 6: 448-457 (October 1981).

8. Clenshaw C. W., G. F. Miller, and M. Woodger, "Algorithms for Special Functions I," Numerische Mathematik, 4: 403-419 (1963).

9. Cody, W. J. "Analysis of Proposals for the Floating-Point Standard." Computer, 14: 63-68 (March 1981).

10. Cody, W. J. and W. Waite. Software Manual for the Elementary Functions. Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1980.

11. Coonen, J. T. "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic," Computer, 13: 68-79 (January 1980).

12. CRC Standard Mathematical Tables (Twenty-fifth Edition), edited by William H. Beyer, Ph.D. Boca Raton, Florida: CRC Press, Inc., 1980.

13. Department of Defense. Reference Manual for the Ada Programming Language, Proposed Standard Document. Arlington, Virginia: Defense Advanced Research Projects Agency, July 1980. (AD A090 709)

14. -----. Requirements for Ada Programming Support Environments: STONEMAN. Washington, D. C.: Department of Defense, February 1980. (AD A100 404)

15. -----. Requirements for High Order Computer Programming Languages: STEELMAN. Washington, D. C.: Department of Defense, June 1978. (AD A059 444)

16. Espinosa, C. Apple II Reference Manual. Cupertino, California: Apple Computer Inc., 1981.

17. Evans, A. A Comparison of Programming Languages: Ada, Praxis, Pascal, C, prepared for Lawrence Livermore National Laboratory. Cambridge, Massachusetts: Bolt Beranek and Newman Inc., April 1981.

18. Fisher, D. A. "DoD's Common Programming Language Effort," Computer, 11: 24-33 (March 1978).

19. Garlington, A. R. Preliminary Design and Implementation of an Ada Pseudo-Machine. MS Thesis. Wright-Patterson AFB, Ohio: Air Force Institute of Technology, March 1981. (AD A100 796)

20. Goodenough, J. B. "The Ada Compiler Validation Capability," Computer, 14: 57-64 (June 1981).

21. Hart, J. F. and others. Computer Approximations. Huntington, New York: Robert E. Krieger Publishing Company, 1978.

22. Honeywell, Inc., Cii Honeywell Bull, and Inria. Formal Definition of the Ada Programming Language. Arlington, Virginia: Defense Advanced Research Projects Agency, November 1980.

23. Hough, D. "Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic," Computer, 14: 70-74 (March 1981).

24. Ichbiah, J. D. and others. "Rationale for the Design of the Ada Programming Language," ACM SIGPLAN Notices, 14: No. 6 Part B (June 1979).

25. Kahan, W. Implementation of Algorithms. Washington, D. C.: Office of Naval Research, 1973. (AD 769 124)

26. Kernighan, B. W. and P. J. Plauger. Software Tools in Pascal. Reading, Massachusetts: Addison-Wesley Publishing Company, 1981.

27. Knuth, D. E. The Art of Computer Programming, Volume 1: Fundamental Algorithms (Second Edition). Reading,

Massachusetts: Addison-Wesley Publishing Company, 1973.

28. -----. The Art of Computer Programming, Volume 2: Seminumerical Algorithms. Reading, Massachusetts: Addison-Wesley Publishing Company, 1971.

29. Loveman, D. B. "Ada: How Big a Difference Will It Make in Software?" Military Electronics/ Countermeasures, 7: 74-84 (May 1981).

30. Miller, G. F. "Algorithms for Special Functions II," Numerische Mathematik, 7: 194-196 (1965).

31. Osborne, A. and J. Kane. An Introduction to Microcomputers, Volume 2: Some Real Microprocessors. Berkeley, California: Osborne & Associates, Inc., 1978.

32. Raskin, J. "Unlimited Precision Division," Byte, 4: 154-156 (February 1979).

33. Softech, Inc. Ada Compiler Validation Implementers' Guide. Arlington, Virginia: Defense Advanced Research Projects Agency, October 1980. (AD A091 760)

34. Softech, Inc. Evaluation of Algol 68, Jovial J3B, Pascal, Simula 67, and Tacpol versus TINMAN Requirements for a Common High Order Programming Language. Waltham, Massachusetts: Softech, Inc., October 1976. (AD A037 637)

35. Stenning, V. and others. "The Ada Environment: A Perspective," Computer, 14: 26-36 (June 1981).

36. Tiberghien, J. The Pascal Handbook. Berkeley, California: Sybex, Inc., 1981.

37. Whitaker, W. A. "Comments on Portions of the ACM SIGPLAN on the Ada Programming Language Not Available in the Proceedings," Ada Implementor's Newsletter. October, January, February, 1981.

38. Wolfe, M. I. and others. "The Ada Language System," Computer, 14: 37-45 (June 1981).

39. Wynn, P. "An Arsenal of Algol Procedures for Complex Arithmetic," BIT, 2: 232-255 (1962).

APPENDIX A

MATHPAK Structure

Support Routines

INITMASKS
POPARG
PUSHARG
WRITEVAL
WRITEBIT
GETSTATZERO
GETNEWSIZE
CHKHIWORDEND
CHKLOWORDEND
SHIFTRIGHT
SHIFTLEFT
BITVALFILL
INTBITREP
RELBITREP
COMPLMENT
ABSVALINT
SETWORDS
EXPOSIZE
CHECKEXPO
SAMEINTPRECISION
SAMERELPRECISION
ADDARGS
BITSTOBITS
FINDNON0BIT
FIND1STFRACBIT
SETSIGBITS
COPYEXPO
POWEROF
CALCEXPO
INTDECREASE (I)
RELDECREASE (I)

Main Functions

Representation

FLOAT
RELTOINT (I)
INTSIZECHANGE
RELSIZECHANGE
ABSVALI (I)
ABSVALR (I)

Arithmetic

PLUSII
PLUSRR
PLUSIR
PLUSRI
PLUSI
PLUSR
MINUSII
MINUSRR
MINUSIR
MINUSRI
MINUSI
MINUSR
MULTII (I)
MULTRR (I)
MULTIR (I)
MULTRI (I)
DIVII (I)
DIVRR (I)
DIVIR (I)
DIVRI (I)
REMI (I)
MODI (I)

Exp/Log

POWERII (I)
POWERRR (I)
POWERIR (I)
POWERRI (I)
SQRTI (I)
SQRTR (I)
EXPI (I)
EXPR (I)
LOGI (I)

LOGR (I)
LNI (I)
LNR (I)

Trig

SINI (I)
SINR (I)
COSI (I)
COSR (I)
TANI (I)
TANR (I)
ARCSINI (I)
ARCSINR (I)
ARCCOSI (I)
ARCCOSR (I)
ARCTANI (I)
ARCTANR (I)
SINHI (I)
SINHR (I)
COSHI (I)
COSHR (I)
TANHI (I)
TANHR (I)

(I) indicates an incomplete module

APPENDIX B

IMPLMENT Structure

Support Routines	Main Functions
SIGNINTADD	INTADD
ADDGENERAL	RELADD
INTPREPADD	INTMULT (I)
CHKHIWRDEND	RELMULT (I)
CHKLOWRDEND	INTDIV (I)
RIGHTSHIFT	RELDIV (I)
LEFTSHIFT	
EXPFROMNUMBER	
EXPSIZE	
ARGTOWORKAREA	
WRKAREATOSTACK	
ZEROOUTEXP	
COMPLEMENT	
EXPTONUMBER	
DECEXP	
CARRYONMSB	
NORMLEFTSHIFT	
ADDNORMALIZE	
RELFRACADDPREP	
RELEXPADDPREP	
PUSHRESULT	
CHKINTADD	
CHKRELADD	
MASKINIT	

(I) indicates an incomplete module

APPENDIX C

MATHPAK Listing

```
(*****************************************************************************)
(*****************************************************************************)
(**                                                                         **)
(**                      P A C K A G E   M A T H P A K                      **)
(**                                                                         **)
(*****************************************************************************)
(*****************************************************************************)


PROCEDURE MATHPAK (FUNCCODE: INTEGER; VAR STAKPTR, ERRCODE: INTEGER);


(*****************************************************************************)
(*****************************************************************************)

(* This is a Pascal Procedure which is meant to correspond to an Ada Package.
The code is written in Pascal, but it is written as much like Ada as possible
so that it can be translated into Ada. *)

(* This Package is designed to receive a function code and a stack pointer
from the caller during execution. The function code tells the package which
function to perform, and the stack pointer tells the package where on the
memory stack to find the arguments for the function. The package "pops" the
arguments from the stack, performs the specified function, and then "pushes"
the result back on the memory stack. It passes back the stack pointer
pointing to the result and an error code. It is assumed that the words of
each argument passed to MATHPAK on the stack run from the most significant
portion of the number (top) to the least significant (bottom). MATHPAK
puts the results back on the stack in the same manner. *)

(* MATHPAK permits arguments of arbitrary precision. It requires the top word
of the stack to contain an integer representing the number of bits of
precision of (number of total bits used by) the first argument. This is
followed by the first argument, starting with the most significant bit of the
stack word and using contiguous words in memory. Then, if two arguments
are necessary for the desired function, the next word contains another integer
for the precision of the second argument, followed by the second argument in
contiguous words of memory. MATHPAK will accept any integer for the given
precisions, but it will use no less than one full word and no more than the
maximum number of words allowed by its implementation. (Note: if an argument
is real, this precision includes the total bits of the sign, exponent, and
fractional part of the number.) It will also round any precision not a
multiple of machine words up to the next full word of precision. The result
will be returned on the top of the stack in the highest precision of the
arguments used rounded up to the next complete machine word. *)
```

```
(* One exception to the above is for the case when MATHPAK is called for the
purpose of changing the precision of an argument.  In this case, the first
word on the stack must contain an integer representing the number of bits
of precision desired for the argument.  The second word must then contain
the present precision of the argument, and then this is followed by the
argument, using contiguous words in the stack.  The result is returned in
the desired precision on the top of the stack. *)

(* The calling routine can either keep track of the precision it expects the
MATHPAK result to have or else it can keep track of the stack position where
it began to load information for the call to MATHPAK.  In the latter case,
both the starting and ending locations of the result on the stack are known,
so the precision can be easily calculated. *)

(* The number representations used by the MATHPAK are as follows:
     Integers are in 2's complement form.
     Floating point (real) numbers have a sign bit, followed by the exponent,
followed by the fractional part.  The exponent uses the smallest number of
bits which can hold an exponent equal to from +4 to -4 times the length of
the fractional part in bits.  The exponent is biased by one less than 2 to
the power of one less than the number of bits in the exponent.  This bias
means that the value stored in the exponent is never negative, so the exponent
does not need a sign.  The fractional part of the real number always has a
positive representation and the binary point is always assumed to be in
front of (to the left of) the first fractional bit.  A non-zero normalized
real number will always have a 1 in the first (left-most) fractional bit. *)

(* In order for calling programs to be able to use MATHPAK, numbers must be
passed to it in the expected representations.  It is usually the
responsibility of the input/output routines of the language, which translate
to and from numerical characters and words of the given numerical type
representing the proper numerical value, to use the proper representations
for the given machine.  When using MATHPAK, an implementation will probably
want to provide modified input/output routines which will work with the
proper MATHPAK representations.  These routines can use MATHPAK's
representation functions, as necessary, to help to accomplish this. *)

(* In order to use the MATHPAK, another package, called IMPLMENT, is also
required.  This package is completely independent of the MATHPAK, but it
contains routines which complete the jobs started by many of the MATHPAK
functions.  IMPLMENT contains routines which can be tailored to a specific
machine implementation, if desired.  The IMPLMENT provided with the MATHPAK
is one which works for the APPLE II, but when completed it should be
implementation independent.  However, no claim is made that it is the most
efficient package for any implementation. *)
```

```
(* MATHPAK's code is designed to be entirely implementation independent.
This is accomplished through the use of the declared constants. When
implementing MATHPAK, before compiling it the constants must be set to the
proper values for the environment. And that's all. It should then work
properly for any implementation, provided the conventions established by
the implementation (through setting the constants) are adhered to and the
proper interfacing (with calling parameters and number representations) is
done. *)

(* The stackpointer used by the MATHPAK is an index into an integer array
representing the memory stack. The calling program must declare this array
and call it "S". The stack is assumed to run from a higher memory location
(bottom) to a lower memory location (top). The constant values for UPSTACK
and DOWNSTACK must be reversed if calling routines are expected to build the
stack in the reverse manner. *)

(* Word bits are indexed assuming the most significant bit is the largest
numbered bit and the least significant bit is the smallest numbered bit.
The constant values for MORESIGBIT and LESSSIGBIT must be reversed if using
this on a computer which indexes bits in the reverse manner. *)

(* In the development of this MATHPAK, the intention has been to use ISO
Standard Pascal throughout. However, the development was done on an APPLE II,
which uses UCSD Pascal. No guarantee can be made that no non-standard Pascal
exists here nor that this MATHPAK will work on every Pascal compiler as written.
In fact, it is probable that some of the bit manipulations will not work
properly in Standard Pascal in their present form. Although some masking has
been introduced in this version, it is not necessarily complete, and packing
and unpacking of packed bit arrays have not even been addressed. Since
UCSD Pascal does automatic packing and unpacking, these issues are of no
concern for an APPLE implementation. However, they make a difference in
Standard Pascal. *)

(* The MATHPAK is presently incomplete. Every possible attempt has been
made to make the documentation reflect the current status and be accurate
and complete for that status. *)
```

```
(*****************************************************************************)
(*****************************************************************************)


CONST                       (* Constants must be set to correspond to the  *)
                            (*                                environment  *)
    WORDSIZE = 16;          (* Number of bits in a machine word used to    *)
                            (*                  perform integer operations  *)
    MSBITNO = 15;           (* The number of the most significant bit of   *)
                            (*                                an integer word  *)
    LSBITNO = 0;            (* The number of the least significant bit of  *)
                            (*                                an integer word  *)
    XTRABIT = 16;           (* One more than the largest legal bit number  *)
                            (*                             of an integer word  *)
    MINBIT = -1;            (* One less than the smallest legal bit number *)
                            (*                             of an integer word  *)
    UPSTACK = -1;           (* -1 if stack built from high memory (bottom) *)
                            (* to low memory (top) and +1 if stack built   *)
                            (* opposite way                                *)
    DOWNSTACK = 1;          (* +1 if stack built from high memory (bottom) *)
                            (* to low memory (top) and -1 if stack built   *)
                            (* opposit way                                 *)
    MORESIGBIT = 1;         (* +1 if bits in a word are numbered from least*)
                            (* significant (lowest number) to most signifi-*)
                            (* cant (highest number) and -1 if bits        *)
                            (* numbered opposite way                       *)
    LESSSIGBIT = -1;        (* -1 if bits in a word are numbered from least*)
                            (* significant (lowest number) to most signifi-*)
                            (* cant (highest number) and +1 if bits        *)
                            (* numbered opposite way                       *)
    MAXWORDSFORARG = 16;    (* Largest number of words permitted for a     *)
                            (* numerical argument for this implementation  *)
    XTRAWORD = 17;          (* One more than MAXWORDSFORARG                *)
```

```
TYPE
    ERRMESSAGE = (NOERROR,CONSTRAINTERROR,SIGNIFICANCELOST,OVERFLOW,UNDERFLOW,
        DIVISIONBYZERO,STORAGEERROR);
    MAXBITINDEX = MINBIT..XTRABIT;
    MAXWORDINDEX = 0..XTRAWORD;
    TRIREP = (A,B,C);
    BITINDEX = LSBITNO..MSBITNO;
    WORDINDEX = 0..MAXWORDSFORARG;
    ARGINDEX = 1..3;
    INTARRAY = ARRAY[ARGINDEX,WORDINDEX] OF INTEGER;
    BITARRAY = ARRAY[ARGINDEX,WORDINDEX] OF PACKED ARRAY[BITINDEX] OF BOOLEAN;
    BOOLARRAY = ARRAY[ARGINDEX,WORDINDEX] OF BOOLEAN;
    ARGUMENT = RECORD CASE TRIREP OF
      A: (INT: INTARRAY);              (* This representation lets MATHPAK   *)
      B: (BIT: BITARRAY);              (* interpret a word any way it wants. *)
      C: (BOOL: BOOLARRAY);            (* This is useful for bit manipula-   *)
      END;                             (* tions.                             *)
    MASKINTARRAY = ARRAY[BITINDEX] OF INTEGER;
    MASKBOOLARRAY = ARRAY[BITINDEX] OF BOOLEAN;
    MASK = RECORD CASE BOOLEAN OF
      TRUE: (INT: MASKINTARRAY);
      FALSE: (BOOL: MASKBOOLARRAY);
      END;

VAR
    BITNO,                                (* Indexes for bit                  *)
    EXPBITNO,                             (*                                  *)
    FLOATBITNO,                           (*                                  *)
    INTBITNO: MAXBITINDEX;                (*                                  *)
    SHIFTBITS,                            (*                                  *)
    BITCOUNT,                             (*                                  *)
    INTBITCOUNT,                          (*                                  *)
    PRECISION,                            (*                                  *)
    EXPOLENGTH,                           (*                                  *)
    EXPOWORDBITS: INTEGER;                (*                                  *)
    WORDNO,                               (*              word                *)
    WORDSUSED,                            (*                                  *)
    WORDDIFF,                             (*                                  *)
    WORDCOUNT: WORDINDEX;                 (*                                  *)
    EXPWORDNO,                            (*                                  *)
    INTWORDNO,                            (*                                  *)
    FLOATWORDNO: MAXWORDINDEX;            (*                                  *)
    ARGNO: ARGINDEX;                      (*                 argument numbers *)
    COUNTER,                              (* General counter *)
    INTBASE,                              (* Variables for                    *)
    EXPBIAS,                              (*            exponent              *)
    EXP: INTEGER;                         (*                   calculations   *)
    ERR: ERRMESSAGE;                      (*     Error message                *)
    ARG: ARGUMENT;                        (*     Working area                 *)
    WORDS: ARRAY [1..3] OF INTEGER;       (* Number of words in each ARG      *)
    MASKARRAY: MASK;          (* Bit masks for changing bit representations *)
```

```
(*****************************************************************************)
(*****************************************************************************)


(*****************************************************************************)
(*                                                                           *)
(*                          SUPPORT ROUTINES                                 *)
(*                                                                           *)
(*****************************************************************************)


    PROCEDURE INITMASKS;      (* Initializes the array of mask words for use *)
                              (* by the routines which change the bit values *)
                              (* of any argument.  This array consists of one*)
                              (* word for each bit in a memory word, and each*)
                              (* mask word contains a 1 in the bit number    *)
                              (* which corresponds to the word number in the *)
                              (* array.  All other bits in each word are 0.  *)
                              (* Uses LSBITNO, BITNO, MSBITNO, MASKARRAY,    *)
                              (* MORESIGBIT, COUNTER, WORDSIZE, LESSSIGBIT.  *)
      BEGIN
        MASKARRAY.INT[LSBITNO] := 1;
        BITNO := LSBITNO + MORESIGBIT;
        FOR COUNTER := 2 TO WORDSIZE DO BEGIN
            MASKARRAY.INT[BITNO] := 2 * MASKARRAY.INT[BITNO+LESSSIGBIT];
            BITNO := BITNO + MORESIGBIT;
        END; (* FOR *)
      END; (* INITMASKS *)

                    (* * * * * * * * * *)


    PROCEDURE POPARG(ARGNUM: ARGINDEX); (* Takes an argument from the memory*)
                                        (* stack and writes it into the     *)
                                        (* appropriate ARG, depending on the*)
                                        (* value of ARGNUM. Leaves the stack*)
                                        (* pointer pointing to the next word*)
                                        (* after the argument on the stack. *)
                                        (* Uses WORDNO, WORDSUSED, ARG, S,  *)
                                        (* STAKPTR, DOWNSTACK.              *)
      BEGIN
        FOR WORDNO := 1 TO WORDSUSED DO BEGIN
            ARG.INT[ARGNUM,WORDNO] := S[STAKPTR];
            STAKPTR := STAKPTR + DOWNSTACK;
        END; (* FOR *)
      END; (* POPARG *)

                    (* * * * * * * * * *)
```

```
PROCEDURE PUSHARG(ARGNUM: ARGINDEX);(* Moves the stack pointer to the   *)
                                    (* next word before the top of the  *)
                                    (* stack and writes the appropriate *)
                                    (* argument from ARG, depending on  *)
                                    (* the value of ARGNUM, onto the top*)
                                    (* of the stack.  Leaves the stack  *)
                                    (* pointer pointing to the new top  *)
                                    (* of the stack.                    *)
                                    (* Uses STAKPTR, WORDNO, WORDSUSED, *)
                                    (* S, ARG, UPSTACK.                 *)
  BEGIN
    FOR WORDNO := WORDSUSED DOWNTO 1 DO BEGIN
        STAKPTR := STAKPTR + UPSTACK;
        S[STAKPTR] := ARG.INT[ARGNUM,WORDNO];
    END; (* FOR *)
  END; (* PUSHARG *)

                    (* * * * * * * * * *)

PROCEDURE WRITEVAL(BITVALUE:        (* Writes a boolean value specified*)
    BOOLEAN; ARGNORES: ARGINDEX;    (* by the parameter BITVALUE to a  *)
    WORDNORES: WORDINDEX;           (* particular bit in ARG specified *)
    BITNORES: BITINDEX);            (* by the index parameters of the  *)
                                    (* call.                           *)
                                    (* Uses ARG, MASKARRAY.            *)
  BEGIN
    IF (BITVALUE = TRUE) THEN
        ARG.BOOL[ARGNORES,WORDNORES] := ARG.BOOL[ARGNORES,WORDNORES] OR
            MASKARRAY.BOOL[BITNORES]
    ELSE
        ARG.BOOL[ARGNORES,WORDNORES] := ARG.BOOL[ARGNORES,WORDNORES] AND
            NOT MASKARRAY.BOOL[BITNORES];
  END; (* WRITEVAL *)

                    (* * * * * * * * * *)
```

```
PROCEDURE WRITEBIT(ARGNOSOURCE:        (* Writes one bit value from one *)
    ARGINDEX; WORDNOSOURCE: WORDINDEX;(* location to another without   *)
    BITNOSOURCE: BITINDEX;ARGNORES:    (* altering the value of any     *)
    ARGINDEX;WORDNORES: WORDINDEX;     (* other bits in the word being  *)
    BITNORES: BITINDEX);               (* written to.  Both the source  *)
                                       (* and result bits must be       *)
                                       (* somewhere in ARG.  The first  *)
                                       (* 3 parameters are indexes for  *)
                                       (* the source bit and the last 3 *)
                                       (* are indexes for the result    *)
                                       (* bit.                          *)
                                       (* Uses ARG.                     *)
                                       (* Calls WRITEVAL.               *)
                                       (* Possible BUGS - ARG will need *)
                                       (* to be unpacked or masked in   *)
                                       (* Standard Pascal to load       *)
                                       (* BOOLVAL properly.             *)
  VAR
    BOOLVAL: BOOLEAN;
  BEGIN
    BOOLVAL := ARG.BIT[ARGNOSOURCE,WORDNOSOURCE,BITNOSOURCE];
    WRITEVAL(BOOLVAL,ARGNORES,WORDNORES,BITNORES);
  END; (* WRITEBIT *)

                    (* * * * * * * * * *)

FUNCTION GETSTATZERO: BOOLEAN; (* Looks at the ARG referenced by the   *)
                               (* present value of ARGNO and checks    *)
                               (* one by one to see if each word of ARG*)
                               (* has a value of zero.  Returns TRUE   *)
                               (* if all words of ARG are zero and     *)
                               (* FALSE if at least one is not.        *)
                               (* Uses COUNTER, WORDS, ARGNO, ARG.     *)
  BEGIN
    COUNTER := 1;
    WHILE ((COUNTER <= WORDS[ARGNO]) AND (ARG.INT[ARGNO,COUNTER] = 0)) DO
        COUNTER := COUNTER + 1;
    IF (COUNTER > WORDS[ARGNO]) THEN
        GETSTATZERO := TRUE
    ELSE
        GETSTATZERO := FALSE;
  END; (* GETSTATZERO *)

                    (* * * * * * * * * *)
```

```
PROCEDURE GETNEWSIZE;     (* Gets the value of the number of bits for  *)
                          (* the new size of the argument from the top *)
                          (* of the stack.  Sets the stack pointer for *)
                          (* the new top of the stack.  Sets WORDS[2]  *)
                          (* equal to the value of the number of words *)
                          (* required to hold the new size of the      *)
                          (* argument.                                 *)
                          (* Uses S, STAKPTR, WORDS, WORDSIZE,         *)
                          (* DOWNSTACK.                                *)
  VAR
    NEWSIZE: INTEGER;
  BEGIN
    NEWSIZE := S[STAKPTR];
    STAKPTR := STAKPTR + DOWNSTACK;
    WORDS[2] := NEWSIZE DIV WORDSIZE;
    IF ((NEWSIZE MOD WORDSIZE) <> 0) THEN
        WORDS[2] := WORDS[2] + 1;
  END; (* GETNEWSIZE *)

                        (* * * * * * * * * * *)

PROCEDURE CHKHIWORDEND(VAR WORDNUM: (* Increments the value of the bit *)
    WORDINDEX; VAR BITNUM:          (* index, BITNUM, to the next more *)
    MAXBITINDEX);                   (* significant bit, and then checks*)
                                    (* to see if its value is past a   *)
                                    (* word boundary.  If so, it decre-*)
                                    (* ments the word number, WORDNUM, *)
                                    (* and sets BITNUM back to the     *)
                                    (* value of the least significant  *)
                                    (* bit of a word, LSBITNO.         *)
                                    (* Uses MSBITNO, LSBITNO,          *)
                                    (* MORESIGBIT.                     *)
  BEGIN
    BITNUM := BITNUM + MORESIGBIT;
    IF (((BITNUM > MSBITNO) AND (MORESIGBIT = 1)) OR ((BITNUM < MSBITNO)
        AND (MORESIGBIT = -1))) THEN BEGIN
        BITNUM := LSBITNO;
        WORDNUM := WORDNUM - 1;
    END; (* IF *)
  END; (* CHKHIWORDEND *)

                    (* * * * * * * * * * *)
```

```
PROCEDURE CHKLOWORDEND(VAR WORDNUM: (* Decrements the value of the bit *)
    MAXWORDINDEX; VAR BITNUM:       (* index, BITNUM, to the next less *)
    MAXBITINDEX);                   (* significant bit, and then checks*)
                                    (* to see if its value is past a   *)
                                    (* word boundary.  If so, it incre-*)
                                    (* ments the word number, WORDNUM, *)
                                    (* and sets BITNUM back to the     *)
                                    (* value of the most significant   *)
                                    (* bit of a word, MSBITNO.         *)
                                    (* Uses LSBITNO, MSBITNO,          *)
                                    (* LESSSIGBIT.                     *)
  BEGIN
    BITNUM := BITNUM + LESSSIGBIT;
    IF (((BITNUM < LSBITNO) AND (LESSSIGBIT = -1)) OR ((BITNUM > LSBITNO)
        AND (LESSSIGBIT = 1))) THEN BEGIN
        BITNUM := MSBITNO;
        WORDNUM := WORDNUM + 1;
    END; (* IF *)
  END; (* CHKLOWORDEND *)

                    (* * * * * * * * * *)

PROCEDURE SHIFTRIGHT(ARGNUM:    (* Shifts bits in the ARG specified by  *)
    ARGINDEX; OLDWORD:          (* ARGNUM.  Writes the bit indexed by   *)
    MAXWORDINDEX; OLDBIT:       (* OLDWORD and OLDBIT to NEWWORD and    *)
    MAXBITINDEX; VAR NEWWORD:   (* NEWBIT.  OLDWORD and OLDBIT must     *)
    MAXWORDINDEX; VAR NEWBIT:   (* begin at the index values for the    *)
    MAXBITINDEX);               (* least significant bit to be shifted, *)
                                (* and NEWWORD and NEWBIT begin at the  *)
                                (* index values for the least signifi-  *)
                                (* cant destination bit.  Works from the*)
                                (* least significant bit up and shifts  *)
                                (* the number of bits specified by      *)
                                (* SHIFTBITS.  Changes index values as  *)
                                (* it goes, and returns the final value *)
                                (* of NEWWORD and NEWBIT.               *)
                                (* Uses SHIFTBITS.                      *)
                                (* Calls WRITEBIT, CHKHIWORDEND.        *)
  BEGIN
    WHILE (SHIFTBITS > 0) DO BEGIN
        WRITEBIT(ARGNUM,OLDWORD,OLDBIT,ARGNUM,NEWWORD,NEWBIT);
        SHIFTBITS := SHIFTBITS - 1;
        CHKHIWORDEND(OLDWORD,OLDBIT);
        CHKHIWORDEND(NEWWORD,NEWBIT);
    END; (* WHILE *)
  END; (* SHIFTRIGHT *)

                    (* * * * * * * * * *)
```

```
PROCEDURE SHIFTLEFT(ARGNUM:   (* Shifts bits in the ARG specified by     *)
    ARGINDEX; VAR NEWWORD:    (* ARGNUM.  Writes the bit indexed by      *)
    MAXWORDINDEX; VAR NEWBIT:(* OLDWORD and OLDBIT to NEWWORD and       *)
    MAXBITINDEX; OLDWORD:     (* NEWBIT.  OLDWORD and OLDBIT must begin  *)
    MAXWORDINDEX; OLDBIT:     (* at the index value for the most signifi-*)
    MAXBITINDEX);             (* cant bit to be shifted, and NEWWORD and *)
                              (* NEWBIT begin at the index value for the *)
                              (* most significant destination bit.  Works*)
                              (* from the most significant bit down and  *)
                              (* shifts the number of bits specified by  *)
                              (* SHIFTBITS.  Changes index values as it  *)
                              (* goes, and returns the final value of    *)
                              (* NEWWORD and NEWBIT.                     *)
                              (* Uses SHIFTBITS.                         *)
                              (* Calls WRITEBIT, CHKLOWORDEND.           *)
  BEGIN
    WHILE (SHIFTBITS > 0) DO BEGIN
        WRITEBIT(ARGNUM,OLDWORD,OLDBIT,ARGNUM,NEWWORD,NEWBIT);
        SHIFTBITS := SHIFTBITS - 1;
        CHKLOWORDEND(OLDWORD,OLDBIT);
        CHKLOWORDEND(NEWWORD,NEWBIT);
    END; (* WHILE *)
  END; (* SHIFTLEFT *)

                    (* * * * * * * * * *)

PROCEDURE BITVALFILL(BOOLVAL: BOOLEAN;(* Fills the bits of the ARG      *)
    ARGNUM: ARGINDEX; WORDNUM:        (* indexed by ARGNUM with the    *)
    MAXWORDINDEX; BITNUM: MAXBITINDEX;(* value of BOOLVAL one by one   *)
    ENDWORD: WORDINDEX; ENDBIT:       (* starting with the bit indexed *)
    BITINDEX);                        (* by WORDNUM and BITNUM and end-*)
                                      (* ing with the bit indexed by   *)
                                      (* ENDWORD and ENDBIT, working   *)
                                      (* from more to less significant *)
                                      (* bits.                         *)
                                      (* Uses LESSSIGBIT.              *)
                                      (* Calls WRITEVAL, CHKLOWORDEND. *)
  BEGIN
    REPEAT
        WRITEVAL(BOOLVAL,ARGNUM,WORDNUM,BITNUM);
        CHKLOWORDEND(WORDNUM,BITNUM);
    UNTIL ((WORDNUM > ENDWORD) OR ((WORDNUM = ENDWORD) AND (((BITNUM <
        ENDBIT) AND (LESSSIGBIT = -1)) OR ((BITNUM > ENDBIT) AND
        (LESSSIGBIT = 1)))));
  END; (* BITVALFILL *)

                    (* * * * * * * * * *)
```

```
PROCEDURE INTBITREP(ARGNUM: ARGINDEX); (* Gets the word containing the  *)
                                       (* precision of the next argument*)
                                       (* from the top of the stack.    *)
                                       (* From this it determines how   *)
                                       (* many words are necessary to   *)
                                       (* hold the next argument as well*)
                                       (* as how many bits it will need *)
                                       (* to be shifted (if any) in     *)
                                       (* order for the argument to end *)
                                       (* on a word boundary.  The      *)
                                       (* argument is then popped from  *)
                                       (* the stack into the appropriate*)
                                       (* ARG as specified by ARGNUM and*)
                                       (* then shifted, if necessary.   *)
                                       (* When bits are shifted, the    *)
                                       (* remaining bits are filled with*)
                                       (* the value of the sign.        *)
                                       (* Uses WORDSUSED, S, STAKPTR,   *)
                                       (* WORDSIZE, SHIFTBITS, WORDS,   *)
                                       (* ARGNO, LSBITNO, ARG, MSBITNO, *)
                                       (* DOWNSTACK, MORESIGBIT,        *)
                                       (* LESSSIGBIT.                   *)
                                       (* Calls SHIFTRIGHT, POPARG,     *)
                                       (* BITVALFILL.                   *)
                                       (* Possible BUGS - ARG will need *)
                                       (* to be unpacked or masked in   *)
                                       (* Standard Pascal to pass the   *)
                                       (* proper value to BITVALFILL.   *)
  VAR
    WORDNEW: MAXWORDINDEX;
    BITOLD, BITNEW: MAXBITINDEX;
  BEGIN
    ARGNO := ARGNUM;
    WORDSUSED := S[STAKPTR] DIV WORDSIZE;
    SHIFTBITS := S[STAKPTR] MOD WORDSIZE;
    STAKPTR := STAKPTR + DOWNSTACK;
    IF (SHIFTBITS <> 0) THEN BEGIN
        WORDSUSED := WORDSUSED + 1;
        WORDNEW := WORDSUSED;
        BITNEW := LSBITNO;
        IF (MORESIGBIT = 1) THEN
            BITOLD := WORDSIZE - SHIFTBITS + LSBITNO
        ELSE
            BITOLD := MSBITNO + SHIFTBITS -1;
        SHIFTBITS := ((WORDSUSED - 1) * WORDSIZE) + SHIFTBITS - 1;
        POPARG(ARGNUM);
        SHIFTRIGHT(ARGNUM,WORDSUSED,BITOLD,WORDNEW,BITNEW);
        CHKHIWORDEND(WORDNEW,BITNEW);
        BITVALFILL(ARG.BIT[ARGNUM,1,MSBITNO],ARGNUM,1,MSBITNO+LESSSIGBIT,
            WORDNEW,BITNEW);
    END (* IF *)
    ELSE
        POPARG(ARGNUM);
```

```
  WORDS[ARGNUM] := WORDSUSED;
END; (* INTBITREP *)

                    (* * * * * * * * * *)

PROCEDURE RELBITREP(ARGNUM: ARGINDEX);(* Gets the word containing the   *)
                                      (* precision of the next argument *)
                                      (* from the top of the stack.     *)
                                      (* From this it determines how    *)
                                      (* many words are necessary to    *)
                                      (* hold the next argument as well *)
                                      (* as how many bits must be added *)
                                      (* (if any) in order to make the  *)
                                      (* argument end on a word         *)
                                      (* boundary.  If bits must be     *)
                                      (* added, it reads the last word  *)
                                      (* of the argument into the first *)
                                      (* word of ARG[3] and writes      *)
                                      (* zeroes on any of the added     *)
                                      (* bits.  Then it writes that     *)
                                      (* last word back to the stack and*)
                                      (* pops the argument from the     *)
                                      (* stack into the appropriate ARG *)
                                      (* as specified by ARGNUM.        *)
                                      (* Uses WORDSUSED, S, STAKPTR,    *)
                                      (* WORDSIZE, SHIFTBITS, WORDS,    *)
                                      (* ARGNO, ARG, BITNO, DOWNSTACK,  *)
                                      (* UPSTACK, LSBITNO, COUNTER,     *)
                                      (* MORESIGBIT.                    *)
                                      (* Calls WRITEVAL, POPARG.        *)
BEGIN
  ARGNO := ARGNUM;
  WORDSUSED := S[STAKPTR] DIV WORDSIZE;
  SHIFTBITS := S[STAKPTR] MOD WORDSIZE;
  STAKPTR := STAKPTR + DOWNSTACK;
  IF (SHIFTBITS <> 0) THEN BEGIN
      WORDSUSED := WORDSUSED + 1;
      ARG.INT[3,1] := S[STAKPTR+WORDSUSED*DOWNSTACK+UPSTACK];
      BITNO := LSBITNO;
      FOR COUNTER := 1 TO (WORDSIZE-SHIFTBITS+LSBITNO) DO BEGIN
          WRITEVAL(FALSE,3,1,BITNO);
          BITNO := BITNO + MORESIGBIT;
      END; (* FOR *)
      S[STAKPTR+WORDSUSED*DOWNSTACK+UPSTACK] := ARG.INT[3,1];
  END; (* IF *)
  WORDS[ARGNUM] := WORDSUSED;
  POPARG(ARGNUM);
END; (* RELBITREP *)

                    (* * * * * * * * * *)
```

```
PROCEDURE COMPLMENT(ARGNUM:    (* Complements an integer argument in ARG.*)
    ARGINDEX; NUMWORDS:        (* Does not change a zero argument.       *)
    WORDINDEX);                (* Uses ARG, WORDNO.                      *)
  VAR
    CHANGEWORD: WORDINDEX;
  BEGIN
    CHANGEWORD := NUMWORDS;
    WHILE((ARG.INT[ARGNUM,CHANGEWORD] = 0) AND (CHANGEWORD <> 1)) DO
        CHANGEWORD := CHANGEWORD - 1;
    IF(ARG.INT[ARGNUM,CHANGEWORD] <> 0) THEN BEGIN
        FOR WORDNO := CHANGEWORD DOWNTO 1 DO
            ARG.BOOL[ARGNUM,WORDNO] := NOT ARG.BOOL[ARGNUM,WORDNO];
        ARG.INT[ARGNUM,CHANGEWORD] := ARG.INT[ARGNUM,CHANGEWORD] + 1;
    END; (* IF *)
  END; (* COMPLMENT *)

                    (* * * * * * * * * *)

PROCEDURE ABSVALINT(ARGNUM:    (* Takes any integer argument from ARG and*)
    ARGINDEX; NUMWORDS:        (* turns it into the absolute value of    *)
    WORDINDEX);                (* that argument.                         *)
                               (* Uses ARG.                              *)
                               (* Calls COMPLMENT.                       *)
  BEGIN
    IF (ARG.INT[ARGNUM,1] < 0) THEN
        COMPLMENT(ARGNUM,NUMWORDS);
  END; (* ABSVALINT *)

                    (* * * * * * * * * *)

PROCEDURE SETWORDS;            (* Assumes that the number of words used by *)
                               (* arguments 1 and 2 in ARG are not the     *)
                               (* same.  Sets ARGNO equal to the number of *)
                               (* the argument which needs to have words   *)
                               (* added (to make argument sizes the same). *)
                               (* It also makes sure that WORDSUSED con-   *)
                               (* tains the value of the argument using the*)
                               (* most words.                              *)
                               (* Uses WORDDIFF, WORDS, ARGNO, WORDSUSED,  *)
                               (* ARG.                                     *)
                               (* Calls ABSVALINT.                         *)
  BEGIN
    ARG.INT[3,1] := WORDS[1] - WORDS[2];
    ABSVALINT(3,1);
    WORDDIFF := ARG.INT[3,1];
    IF (WORDS[1] > WORDS[2]) THEN
        ARGNO := 2
    ELSE BEGIN
        ARGNO := 1;
        WORDSUSED := WORDS[2];
    END; (* ELSE *)
  END; (* SETWORDS *)
```

```
                    (* * * * * * * * * *)

FUNCTION EXPOSIZE(NUMOFWORDS:    (* Calculates the proper sized exponent *)
    INTEGER): INTEGER;           (* for a real argument from the number  *)
                                 (* of words in the argument given as the*)
                                 (* calling parameter.  It assumes that  *)
                                 (* the smallest possible word size is 8 *)
                                 (* bits.  It uses an algorithm which    *)
                                 (* determines the smallest exponent     *)
                                 (* length which can hold an exponent    *)
                                 (* equal to from +4 to -4 times the     *)
                                 (* length of the fractional part of the *)
                                 (* real number in bits.  It returns the *)
                                 (* number of bits in the length of this *)
                                 (* smallest exponent as the value of the*)
                                 (* function.                            *)
                                 (* Uses PRECISION, WORDSIZE.            *)
  VAR
    TRIALEXPO,TRIALFRAC,TRIALPRECISION: INTEGER;
  BEGIN
    PRECISION := NUMOFWORDS * WORDSIZE;
    TRIALEXPO := 5;
    TRIALFRAC := 4;
    TRIALPRECISION := TRIALEXPO + TRIALFRAC + 1;
    WHILE (TRIALPRECISION < PRECISION) DO BEGIN
        TRIALEXPO := TRIALEXPO + 1;
        TRIALFRAC := TRIALFRAC * 2;
        TRIALPRECISION := TRIALEXPO + TRIALFRAC + 1;
    END; (* WHILE *)
    EXPOSIZE := TRIALEXPO;
  END; (* EXPOSIZE *)

                    (* * * * * * * * * *)
```

```
PROCEDURE CHECKEXPO;        (* Checks the size of the old exponent in an *)
                            (* argument in ARG specified by ARGNO        *)
                            (* against the size of the exponent required *)
                            (* by an argument with the number of words   *)
                            (* specified in WORDSUSED.  If the exponent  *)
                            (* lengths are not the same, the one         *)
                            (* determined from WORDSUSED is assumed to be*)
                            (* larger, and the argument in ARG gets its  *)
                            (* exponent increased by the number of bits  *)
                            (* necessary to make it equal to that        *)
                            (* determined by WORDSUSED.  The argument in *)
                            (* ARG is now assumed to use the number of   *)
                            (* words specified in WORDSUSED and all its  *)
                            (* bits except the sign bit and the first    *)
                            (* bit of the exponent are shifted to the    *)
                            (* right by the same number of bits which    *)
                            (* were added to the exponent.  The bits     *)
                            (* added to the exponent are the opposite    *)
                            (* value of the first bit of the exponent.   *)
                            (* This serves to adjust the bias for the    *)
                            (* new exponent size.                        *)
                            (* Uses WORDS, ARGNO, WORDSUSED, SHIFTBITS,  *)
                            (* MSBITNO, LSBITNO, WORDSIZE, ARG,          *)
                            (* MORESIGBIT, LESSSIGBIT.                   *)
                            (* Calls EXPOSIZE, SHIFTRIGHT, BITVALFILL.   *)
                            (* Possible BUGS - ARG will need to be un-   *)
                            (* packed or masked in Standard Pascal to get*)
                            (* the proper value for OPPBIT.              *)
  VAR
    WORDOLD, WORDNEW: MAXWORDINDEX;
    BITOLD, BITNEW,
    OLDEXPOSIZE,NEWEXPOSIZE: INTEGER;
    OPPBIT: BOOLEAN;
  BEGIN
    OLDEXPOSIZE := EXPOSIZE(WORDS[ARGNO]);
    NEWEXPOSIZE := EXPOSIZE(WORDSUSED);
    SHIFTBITS := WORDS[ARGNO] * WORDSIZE - 2;
    WORDOLD := WORDS[ARGNO];
    WORDS[ARGNO] := WORDSUSED;
    IF ((NEWEXPOSIZE-OLDEXPOSIZE) > 0) THEN BEGIN
        BITOLD := LSBITNO;
        WORDNEW := WORDOLD + ((OLDEXPOSIZE - NEWEXPOSIZE) DIV WORDSIZE) + 1;
        BITNEW := MSBITNO + (((OLDEXPOSIZE - NEWEXPOSIZE) MOD WORDSIZE)
            * LESSSIGBIT) + MORESIGBIT;
        IF (((BITNEW > MSBITNO) AND (MORESIGBIT = 1)) OR ((BITNEW <
            MSBITNO) AND (MORESIGBIT = -1))) THEN BEGIN
            BITNEW := LSBITNO;
            WORDNEW := WORDNEW - 1;
        END; (* IF *)
        SHIFTRIGHT(ARGNO,WORDOLD,BITOLD,WORDNEW,BITNEW);
        OPPBIT := NOT ARG.BIT[ARGNO,1,MSBITNO+LESSSIGBIT];
        BITVALFILL(OPPBIT,ARGNO,1,MSBITNO+2*LESSSIGBIT,WORDNEW,BITNEW);
    END; (* IF *)
```

```
END; (* CHECKEXPO *)

                (* * * * * * * * * *)

PROCEDURE SAMEINTPRECISION;     (* Checks to see if the two arguments in *)
                                (* ARG have the same precision.  If not  *)
                                (* it takes the one with least precision *)
                                (* and expands it to one with precision  *)
                                (* equal to the other.  It does this by  *)
                                (* moving all words down the appropriate *)
                                (* number of words and then filling the  *)
                                (* words added in front with bits with   *)
                                (* the same value as the sign of the     *)
                                (* argument.  It also makes sure that    *)
                                (* WORDSUSED ends up with the right      *)
                                (* value.                                *)
                                (* Uses WORDSUSED, WORDS, WORDNO,        *)
                                (* WORDCOUNT, ARG, ARGNO, WORDDIFF,      *)
                                (* LSBITNO, MSBITNO, LESSSIGBIT.         *)
                                (* Calls SETWORDS, BITVALFILL.           *)
                                (* Possible BUGS - ARG will need to be   *)
                                (* unpacked or masked in Standard Pascal *)
                                (* to pass the proper value to BITVALFILL*)
  BEGIN
    WORDSUSED := WORDS[1];
    IF (WORDS[1] <> WORDS[2]) THEN BEGIN
        SETWORDS;
        WORDNO := WORDSUSED;
        FOR WORDCOUNT := 1 TO WORDS[ARGNO] DO BEGIN
            ARG.INT[ARGNO,WORDNO] := ARG.INT[ARGNO,WORDNO-WORDDIFF];
            WORDNO := WORDNO - 1;
        END; (* FOR *)
        BITVALFILL(ARG.BIT[ARGNO,1,MSBITNO],ARGNO,1,MSBITNO+LESSSIGBIT,
            WORDDIFF,LSBITNO);
    END; (* IF *)
  END; (* SAMEINTPRECISION *)

                (* * * * * * * * * *)
```

```
PROCEDURE SAMERELPRECISION; (* Checks to see if the two arguments in   *)
                            (* ARG have the same precision.  If not    *)
                            (* it takes the one with least precision   *)
                            (* and expands it to one with the same     *)
                            (* precision as the other.  It does this by*)
                            (* adding the appropriate number of words  *)
                            (* of zero to the end of the argument.  It *)
                            (* then checks the length of the exponent  *)
                            (* against the length it should have for   *)
                            (* the new precision.  If the exponent     *)
                            (* needs to be lengthened it does this by  *)
                            (* adding the appropriate bits to the front*)
                            (* of the exponent and shifting the re-    *)
                            (* maining bits of the exponent and the    *)
                            (* bits of the fractional part the appro-  *)
                            (* riate number of bits to the right.  It  *)
                            (* also makes sure that WORDSUSED and      *)
                            (* EXPOLENGTH end up with the right value. *)
                            (* Uses WORDSUSED, WORDS, WORDNO, ARGNO,   *)
                            (* ARG, EXPOLENGTH.                        *)
                            (* Calls SETWORDS, CHECKEXPO, EXPOSIZE.    *)
  BEGIN
    WORDSUSED := WORDS[1];
    IF (WORDS[1] <> WORDS[2]) THEN BEGIN
        SETWORDS;
        FOR WORDNO := (WORDS[ARGNO]+1) TO WORDSUSED DO
            ARG.INT[ARGNO,WORDNO] := 0;
        CHECKEXPO;
    END; (* IF *)
    EXPOLENGTH := EXPOSIZE(WORDSUSED);
  END; (* SAMERELPRECISION *)

                    (* * * * * * * * * * *)
```

```
PROCEDURE ADDARGS       (* Pushes the arguments in ARG[1] and ARG[2] on  *)
    (ADDTYPE: INTEGER);(* the stack (along with a word specifying the   *)
                       (* word length of the arguments) and calls       *)
                       (* IMPLMENT to add them together.  Uses the      *)
                       (* parameter ADDTYPE to set the code for the type*)
                       (* of add (integer, real, etc.) to pass to       *)
                       (* IMPLMENT.  Then pops the result from the stack*)
                       (* into ARG[1].  It assumes that the arguments   *)
                       (* each occupy the number of words specified in  *)
                       (* WORDSUSED and that both arguments are         *)
                       (* integers.                                     *)
                       (* Uses STAKPTR, S, WORDSUSED, ERRCODE, UPSTACK. *)
                       (* Calls PUSHARG, IMPLMENT, POPARG.              *)
  BEGIN
    PUSHARG(1);
    PUSHARG(2);
    STAKPTR := STAKPTR + UPSTACK;
    S[STAKPTR] := WORDSUSED;
    IMPLMENT(ADDTYPE,STAKPTR,ERRCODE);
    POPARG(1);
  END; (* ADDARGS *)

                    (* * * * * * * * * *)

PROCEDURE BITSTOBITS      (* Writes bits one at a time, in order, from *)
    (NUMOFBITS: INTEGER;  (* most to least significant, from one       *)
    FROMARGNO: ARGINDEX;  (* argument in ARG to another.  It uses the  *)
    FROMWORDNO:           (* argument, word, and bit indexes passed as *)
    MAXWORDINDEX;         (* calling parameters.  It copies the number *)
    FROMBITNO:            (* of bits specified by NUMOFBITS and does   *)
    MAXBITINDEX; TOARGNO: (* not need the bits copied to either begin  *)
    ARGINDEX; VAR         (* or end on a word boundary.  Returns the   *)
    TOWORDNO:             (* final value of TOWORDNO and TOBITNO.      *)
    MAXWORDINDEX; VAR     (* Uses BITCOUNT.                            *)
    TOBITNO: MAXBITINDEX);(* Calls WRITEBIT, CHKLOWORDEND.             *)
  BEGIN
    FOR BITCOUNT := 1 TO NUMOFBITS DO BEGIN
        WRITEBIT(FROMARGNO,FROMWORDNO,FROMBITNO,TOARGNO,TOWORDNO,TOBITNO);
        CHKLOWORDEND(FROMWORDNO,FROMBITNO);
        CHKLOWORDEND(TOWORDNO,TOBITNO);
    END; (* FOR *)
  END; (* BITSTOBITS *)

                    (* * * * * * * * * *)
```

```
PROCEDURE FINDNONOBIT(ARGNUM: (* Finds the first non-zero bit in the    *)
    ARGINDEX);                (* appropriate argument in ARG specified  *)
                              (* by ARGNUM, starting at the word and bit*)
                              (* indexed by INTWORDNO and INTBITNO.  It *)
                              (* goes through the bits from most to     *)
                              (* least significant until a 1 is found.  *)
                              (* INTBITCOUNT is incremented each time   *)
                              (* the indexes are incremented for a new  *)
                              (* bit, and INTWORDNO and INTBITNO remain *)
                              (* pointing to the "found" bit upon       *)
                              (* return.                                *)
                              (* Uses ARG, INTWORDNO, INTBITNO,         *)
                              (* INTBITCOUNT.                           *)
                              (* Calls CHKLOWORDEND.                    *)
                              (* Possible BUGS - ARG will need to be    *)
                              (* unpacked or masked in Standard Pascal  *)
                              (* to make the test for FALSE work        *)
                              (* properly.                              *)
  BEGIN
    WHILE(ARG.BIT[ARGNUM,INTWORDNO,INTBITNO] = FALSE) DO BEGIN
        CHKLOWORDEND(INTWORDNO,INTBITNO);
        INTBITCOUNT := INTBITCOUNT + 1;
    END; (* WHILE *)
  END; (* FINDNONOBIT *)

                (* * * * * * * * * * *)
```

```
PROCEDURE FINDlSTFRACBIT;  (* Finds the first bit of the fractional part*)
                           (* of a real number.  It does this by first  *)
                           (* determining how long the exponent of the  *)
                           (* number is and then comparing this with the*)
                           (* wordsize to make a final determination of *)
                           (* exactly what bit of what word in the      *)
                           (* argument is the first fractional bit.  It *)
                           (* gets the size of the argument from        *)
                           (* WORDSUSED and leaves the word and bit     *)
                           (* indexes for the "found" bit in FLOATWORDNO*)
                           (* and FLOATBITNO respectively.              *)
                           (* Uses EXPOLENGTH, FLOATWORDNO, WORDSIZE,   *)
                           (* EXPOWORDBITS, FLOATBITNO, WORDSUSED,      *)
                           (* MORESIGBIT, LSBITNO, LESSSIGBIT.          *)
                           (* Calls EXPOSIZE.                           *)
  BEGIN
    EXPOLENGTH := EXPOSIZE(WORDSUSED);
    FLOATWORDNO := 1;
    EXPOWORDBITS := WORDSIZE;
    WHILE (EXPOWORDBITS < EXPOLENGTH) DO BEGIN
        EXPOWORDBITS := EXPOWORDBITS + WORDSIZE;
        FLOATWORDNO := FLOATWORDNO + 1;
    END; (* WHILE *)
    FLOATBITNO := (EXPOWORDBITS - EXPOLENGTH) * MORESIGBIT + LSBITNO
        + 2 * LESSSIGBIT;
  END; (* FINDlSTFRACBIT *)

                    (* * * * * * * * * * *)

PROCEDURE SETSIGBITS;      (* Takes the values of WORDSUSED and         *)
                           (* INTBITCOUNT and uses them to determine the *)
                           (* number of significant bits in an integer  *)
                           (* argument which is to be converted to a real.*)
                           (* Then it checks to see if the real number  *)
                           (* will have enough space for its exponent plus*)
                           (* all the significant bits from the integer if*)
                           (* the same precision is used for both.  If not*)
                           (* then INTBITCOUNT is set so that the neces- *)
                           (* sary number of bits of precision will be  *)
                           (* truncated from the real number and a warning*)
                           (* message informs the user that some precision*)
                           (* has been lost.                            *)
                           (* Uses PRECISION, WORDSUSED, WORDSIZE,      *)
                           (* INTBITCOUNT, ERR, EXPOLENGTH.             *)
  BEGIN
    PRECISION := WORDSUSED * WORDSIZE;
    INTBITCOUNT := PRECISION - INTBITCOUNT;
    IF ((INTBITCOUNT + EXPOLENGTH + 1) > PRECISION) THEN BEGIN
            ERR := SIGNIFICANCELOST;
            INTBITCOUNT := PRECISION  - EXPOLENGTH - 1;
    END; (* IF *)
  END; (* SETSIGBITS *)
```

```
                    (* * * * * * * * * *)

PROCEDURE COPYEXPO;     (* Copies an exponent from ARG[3] to the argument*)
                        (* in the ARG specified by ARGNO.  The exponent  *)
                        (* in ARG[3] is expected to begin in word 1 and  *)
                        (* the least significant bit is expected to be   *)
                        (* right justified to a word boundary.  The      *)
                        (* argument specified by ARGNO is a real number  *)
                        (* occupying an integral number of full words.   *)
                        (* Hence the copying begins at the bit following *)
                        (* the sign bit and continues to the end of the  *)
                        (* exponent.  Each bit is copied one at a time.  *)
                        (* Uses EXPWORDNO, EXPBITNO, EXPOLENGTH, MSBITNO,*)
                        (* WORDSIZE, FLOATBITNO, FLOATWORDNO, BITCOUNT,  *)
                        (* ARGNO, MORESIGBIT, LSBITNO, LESSSIGBIT.       *)
                        (* Calls WRITEBIT, CHKLOWORDEND.                 *)
  BEGIN
    EXPWORDNO := 1;
    BITCOUNT := EXPOLENGTH;
    WHILE (BITCOUNT > WORDSIZE) DO BEGIN
        BITCOUNT := BITCOUNT - WORDSIZE;
        EXPWORDNO := EXPWORDNO + 1;
    END; (* WHILE *)
    EXPBITNO := BITCOUNT * MORESIGBIT + LSBITNO + LESSSIGBIT;
    FLOATBITNO := MSBITNO + LESSSIGBIT;
    FLOATWORDNO := 1;
    FOR BITCOUNT := 1 TO EXPOLENGTH DO BEGIN
        WRITEBIT(3,EXPWORDNO,EXPBITNO,ARGNO,FLOATWORDNO,FLOATBITNO);
        CHKLOWORDEND(FLOATWORDNO,FLOATBITNO);
        CHKLOWORDEND(EXPWORDNO,EXPBITNO);
    END; (* FOR *)
  END; (* COPYEXPO *)

                    (* * * * * * * * * *)
```

```
FUNCTION POWEROF(BASE: INTEGER;          (* Takes a base and exponent as    *)
    EXPONENT: INTEGER): INTEGER;         (* calling parameters and performs *)
                                         (* the operation of exponentiation *)
                                         (* by using the base as a factor   *)
                                         (* the exponent number of times.   *)
                                         (* Uses COUNTER.                   *)
                                         (* Known BUGS - only works for non-*)
                                         (* negative integers which can be  *)
                                         (* held in one word.               *)
  VAR
    PRODUCT: INTEGER;
  BEGIN
    PRODUCT := 1;
    COUNTER := 1;
    WHILE (COUNTER <= EXPONENT) DO BEGIN
        PRODUCT := PRODUCT * BASE;
        COUNTER := COUNTER + 1;
    END; (* WHILE *)
    POWEROF := PRODUCT;
  END; (* POWEROF *)

                    (* * * * * * * * * * *)

PROCEDURE CALCEXPO;                      (* Determines the appropriate bias for the *)
                                         (* exponent of a real number based on the  *)
                                         (* number of bits in the exponent.  The    *)
                                         (* algorithm used is 2 to the power of one *)
                                         (* less than the number of bits in the     *)
                                         (* exponent, then subtract 1 from the      *)
                                         (* result.  This determines the exponent   *)
                                         (* bias.  Then the value of the exponent,  *)
                                         (* determined from the value of            *)
                                         (* INTBITCOUNT, is added to the bias,      *)
                                         (* resulting in the value which is actually*)
                                         (* to be stored in the exponent of the real*)
                                         (* number.  This value is written to       *)
                                         (* ARG[3].                                 *)
                                         (* Uses INTBASE, EXP, EXPOLENGTH, EXPBIAS, *)
                                         (* INTBITCOUNT, ARG.                       *)
                                         (* Calls POWEROF.                          *)
                                         (* Known BUGS - only works for values which*)
                                         (* can be held in one word because POWEROF *)
                                         (* works for arguments of only one word.   *)
  BEGIN
    INTBASE := 2;
    EXP := EXPOLENGTH - 1;
    EXPBIAS := POWEROF(INTBASE,EXP) - 1;
    EXP := EXPBIAS + INTBITCOUNT;
    ARG.INT[3,1] := EXP;
  END; (* CALCEXPO *)

              (* * * * * * * * * *)
```

```
PROCEDURE INTDECREASE;
  BEGIN
  END; (* INTDECREASE *)

            (* * * * * * * * * * *)

PROCEDURE RELDECREASE;
  BEGIN
  END; (* RELDECREASE *)
```

```
(*****************************************************************************)
(*****************************************************************************)


(*****************************************************************************)
(*                                                                           *)
(*                          MATHPAK "FUNCTIONS"                              *)
(*                                                                           *)
(*****************************************************************************)


    PROCEDURE FLOAT (ARGNUM: ARGINDEX); (* Changes a number from an integer *)
                                        (* representation to a floating     *)
                                        (* point representation.  It works  *)
                                        (* on an argument which is on the   *)
                                        (* stack (preceded by a word indica-*)
                                        (* ting its precision).  It pops the*)
                                        (* integer argument off the stack   *)
                                        (* and into ARG[3].  If the entire  *)
                                        (* argument is zero, then it makes  *)
                                        (* no change in representation.  If *)
                                        (* not zero, it checks to see if the*)
                                        (* argument is negative and, if so, *)
                                        (* changes all but the sign to the  *)
                                        (* corresponding positive represen- *)
                                        (* tation.  Then it finds the most  *)
                                        (* significant non-zero bit and     *)
                                        (* determines the number of         *)
                                        (* significant bits in the integer. *)
                                        (* It writes these significant bits *)
                                        (* one at a time, from most to least*)
                                        (* significant, into the fractional *)
                                        (* part of the real representation  *)
                                        (* in the ARG specified by ARGNUM.  *)
                                        (* It fills the fraction with zeroes*)
                                        (* to the next word boundary, if    *)
                                        (* necessary.  It writes the        *)
                                        (* appropriate exponent and sign    *)
                                        (* into the real representation to  *)
                                        (* make it complete.  It also sets  *)
                                        (* WORDS for the ARG specified by   *)
                                        (* ARGNUM.                          *)
                                        (* Uses INTWORDNO, FLOATWORDNO,     *)
                                        (* MSBITNO, INTBITNO, INTBITCOUNT,  *)
                                        (* WORDNO, WORDSUSED, ARG, ARGNO,   *)
                                        (* WORDS, FLOATBITNO, LSBITNO,      *)
                                        (* LESSSIGBIT.                      *)
                                        (* Calls INTBITREP, GETSTATZERO,    *)
                                        (* WRITEBIT, FINDNONOBIT, CALCEXPO, *)
                                        (* FIND1STFRACBIT, SETSIGBITS,      *)
                                        (* BITSTOBITS, BITVALFILL, COPYEXPO,*)
                                        (* ABSVALINT.                       *)
```

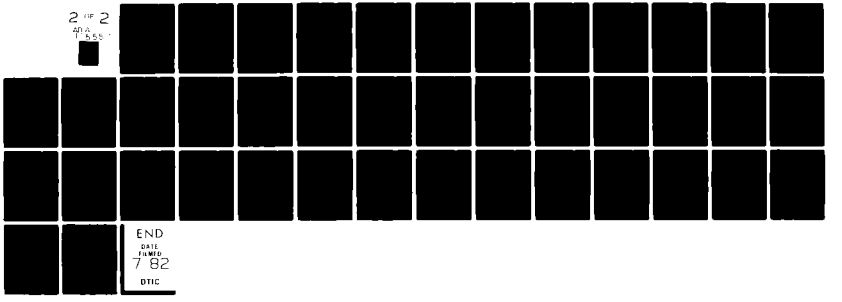
```
  BEGIN
    INTBITREP(3);
    ARGNO := ARGNUM;
    WORDS[ARGNUM] := WORDSUSED;
    IF (GETSTATZERO = TRUE) THEN BEGIN
        FOR WORDNO := 1 TO WORDSUSED DO
            ARG.INT[ARGNUM,WORDNO] := 0;
    END (* IF *)
    ELSE BEGIN
        INTWORDNO := 1;
        FLOATWORDNO := 1;
        WRITEBIT(3,1,MSBITNO,ARGNUM,1,MSBITNO);
        IF (ARG.INT[3,1] < 0) THEN BEGIN
            ABSVALINT(3,WORDS[3]);
        END; (* IF *)
        INTBITNO := MSBITNO + LESSSIGBIT;
        INTBITCOUNT := 1;
        FINDNONOBIT(3);
        FIND1STFRACBIT;
        SETSIGBITS;
        BITSTOBITS(INTBITCOUNT,3,INTWORDNO,INTBITNO,ARGNO,FLOATWORDNO,
            FLOATBITNO);
        BITVALFILL(FALSE,ARGNO,FLOATWORDNO,FLOATBITNO,WORDSUSED,LSBITNO);
        CALCEXPO;
        COPYEXPO;
    END; (* ELSE *)
  END; (* FLOAT *)

                    (* * * * * * * * * *)

PROCEDURE RELTOINT;
  BEGIN
  END; (* RELTOINT *)

                    (* * * * * * * * * *)
```

AD-A115 557 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
ARBITRARY PRECISION IN A PRELIMINARY MATH UNIT FOR ADA.(U)
MAR 82 P K LAWLIS
UNCLASSIFIED AFIT/GCS/MA/82M-2 NL



```
PROCEDURE INTSIZECHANGE;  (* Gets a word containing the precision for   *)
                          (* the new integer argument from the top of   *)
                          (* the stack.  Then gets a word containing the*)
                          (* present precision of the argument from the *)
                          (* next location on the stack.  Then takes the*)
                          (* argument from the stack and places it in   *)
                          (* ARG[1].  Changes its precision so that it  *)
                          (* has the desired new precision (plus up to  *)
                          (* the next word boundary if precision re-    *)
                          (* quested was not an even number of words),  *)
                          (* and leaves it in ARG[1].  Leaves the stack *)
                          (* pointer pointing to the next word after    *)
                          (* the argument on the stack.                 *)
                          (* Uses WORDS.                                *)
                          (* Calls GETNEWSIZE, INTBITREP, INTDECREASE,  *)
                          (* SAMEINTPRECISION.                          *)
  BEGIN
    GETNEWSIZE;
    INTBITREP(1);
    IF (WORDS[2] < WORDS[1]) THEN
        INTDECREASE
    ELSE
        SAMEINTPRECISION;
  END; (* INTSIZECHANGE *)

                (* * * * * * * * * *)

PROCEDURE RELSIZECHANGE;  (* Gets a word containing the precision for   *)
                          (* the new real argument from the top of the  *)
                          (* stack.  Then gets a word containing the    *)
                          (* present precision of the argument from the *)
                          (* next location on the stack.  Then takes the*)
                          (* argument from the stack and places it in   *)
                          (* ARG[1].  Changes its precision so that it  *)
                          (* has the desired new precision (plus up to  *)
                          (* the next word boundary if precision re-    *)
                          (* quested was not an even number of words),  *)
                          (* and leaves it in ARG[1].  Leaves the stack *)
                          (* pointer pointing to the next word after the*)
                          (* argument on the stack.                     *)
                          (* Uses WORDS.                                *)
                          (* Calls GETNEWSIZE, RELBITREP, RELDECREASE,  *)
                          (* SAMERELPRECISION.                          *)
  BEGIN
    GETNEWSIZE;
    RELBITREP(1);
    IF (WORDS[2] < WORDS[1]) THEN
        RELDECREASE
    ELSE
        SAMERELPRECISION;
  END; (* RELSIZECHANGE *)
```

```
                    (* * * * * * * * * *)

PROCEDURE ABSVALI;
  BEGIN
  END; (* ABSVALI *)

                    (* * * * * * * * * *)

PROCEDURE ABSVALR;
  BEGIN
  END; (* ABSVALR *)

                    (* * * * * * * * * *)

PROCEDURE PLUSII;       (* Takes two integer arguments from the stack    *)
                        (* (each preceded by a word indicating its       *)
                        (* precision), adds them together, and leaves the*)
                        (* result of the addition in ARG[1]. The         *)
                        (* arguments are placed in ARG[2] and ARG[1],    *)
                        (* respectively, when popped from the stack.     *)
                        (* A check is also made to see if the two        *)
                        (* arguments have the same precision, and if not,*)
                        (* then the one of least precision has its       *)
                        (* representation increased to match the other.  *)
                        (* The result has the precision of the argument  *)
                        (* with the greatest precision.                  *)
                        (* Uses no variables.                            *)
                        (* Calls INTBITREP, SAMEINTPRECISION, ADDARGS.   *)
  BEGIN
    INTBITREP(2);
    INTBITREP(1);
    SAMEINTPRECISION;
    ADDARGS(1);
  END; (* PLUSII *)

                        (* * * * * * * * * *)
```

```
PROCEDURE PLUSRR;       (* Takes two real arguments from the stack (each *)
                        (* preceded by a word indicating its precison),  *)
                        (* adds them together, and places the result of  *)
                        (* the addition in ARG[1]. The two arguments     *)
                        (* are placed in ARG[2] and ARG[1], respectively,*)
                        (* when popped from the stack. A check is also   *)
                        (* made to see if the arguments have the same    *)
                        (* precision, and if not, then the one of least  *)
                        (* precision has its representation increased to *)
                        (* match the other. The result has the precision*)
                        (* of the argument with the greatest precison.   *)
                        (* Uses no variables.                            *)
                        (* Calls RELBITREP, SAMERELPRECISION, ADDARGS.   *)
  BEGIN
    RELBITREP(2);
    RELBITREP(1);
    SAMERELPRECISION;
    ADDARGS(2);
  END; (* PLUSRR *)

                        (* * * * * * * * * * *)

PROCEDURE PLUSIR;        (* First takes a real argument from the top of  *)
                         (* the stack (preceded by a word indicating its *)
                         (* precision), and reads it into ARG[2]. Then   *)
                         (* it takes an integer argument from the top of *)
                         (* stack (also preceded by a word indicating its*)
                         (* precision), changes its representation to    *)
                         (* floating point, and puts it in ARG[1]. Then  *)
                         (* it checks to see if the arguments have the   *)
                         (* same precision, and if not, then the one of  *)
                         (* least precision has its precision increased  *)
                         (* to match the other. The arguments are then   *)
                         (* added together. The result is a floating     *)
                         (* argument in ARG[1] with a precision the same *)
                         (* as the argument with greatest precision.     *)
                         (* Uses no variables.                           *)
                         (* Calls RELBITREP, FLOAT, SAMERELPRECISION,    *)
                         (* ADDARGS.                                     *)
  BEGIN
    RELBITREP(2);
    FLOAT(1);
    SAMERELPRECISION;
    ADDARGS(2);
  END; (* PLUSIR *)

                         (* * * * * * * * * * *)
```

```
PROCEDURE PLUSRI;       (* First takes an integer argument from the top  *)
                        (* of the stack (preceded by a word indicating   *)
                        (* its precision), changes its representation to *)
                        (* floating point, and puts it in ARG[2]. Then   *)
                        (* it takes a real argument from the top of the  *)
                        (* stack (also preceded by a word indicating its *)
                        (* precision) and puts it in ARG[1]. Then it     *)
                        (* checks to see if the arguments have the same  *)
                        (* precision, and if not, it changes the         *)
                        (* representation of the argument of least       *)
                        (* precision to make its precision match the     *)
                        (* other. Then it adds the arguments together.   *)
                        (* The result is a floating argument in ARG[1]   *)
                        (* with the same precision as the argument of    *)
                        (* greatest precision.                           *)
                        (* Uses no variables.                            *)
                        (* Calls FLOAT, RELBITREP, SAMERELPRECISION,     *)
                        (* ADDARGS.                                      *)
  BEGIN
    FLOAT(2);
    RELBITREP(1);
    SAMERELPRECISION;
    ADDARGS(2);
  END; (* PLUSRI *)

                    (* * * * * * * * * *)

PROCEDURE PLUSI;        (* Takes an integer argument from the top of the *)
                        (* stack (preceded by a word indicating its      *)
                        (* precision) and puts it in ARG[1].             *)
                        (* Uses no variables.                            *)
                        (* Calls INTBITREP.                              *)
  BEGIN
    INTBITREP(1);
  END; (* PLUSI *)

                    (* * * * * * * * * *)

PROCEDURE PLUSR;        (* Takes a real argument from the top of the     *)
                        (* stack (preceded by a word indicating its      *)
                        (* precision) and puts it in ARG[1].             *)
                        (* Uses no variables.                            *)
                        (* Calls RELBITREP.                              *)
  BEGIN
    RELBITREP(1);
  END; (* PLUSR *)

                    (* * * * * * * * * *)
```

```
PROCEDURE MINUSII;      (* Same operation as PLUSII except that    *)
                        (* ARG[2] is complemented before the       *)
                        (* addition is performed.                  *)
                        (* Uses WORDS.                             *)
                        (* Calls INTBITREP, COMPLMENT, ADDARGS,    *)
                        (* SAMEINTPRECISION.                       *)
  BEGIN
    INTBITREP(2);
    COMPLMENT(2,WORDS[2]);
    INTBITREP(1);
    SAMEINTPRECISION;
    ADDARGS(1);
  END; (* MINUSII *)

                    (* * * * * * * * * *)

PROCEDURE MINUSRR;      (* Same operation as PLUSRR except that the *)
                        (* sign of ARG[2] is changed before the    *)
                        (* addition is performed.                  *)
                        (* Uses ARG, MSBITNO.                      *)
                        (* Calls RELBITREP, SAMERELPRECISION,      *)
                        (* ADDARGS.                                *)
                        (* Possible BUGS - ARG will need to be     *)
                        (* unpacked or masked in Standard Pascal to *)
                        (* make the NOT work properly.             *)
  BEGIN
    RELBITREP(2);
    ARG.BIT[2,1,MSBITNO] := NOT ARG.BIT[2,1,MSBITNO];
    RELBITREP(1);
    SAMERELPRECISION;
    ADDARGS(2);
  END; (* MINUSRR *)

                    (* * * * * * * * * *)

PROCEDURE MINUSIR;      (* Same operation as PLUSIR except that the  *)
                        (* sign of ARG[2] is changed before the    *)
                        (* addition is performed.                  *)
                        (* Uses ARG, MSBITNO.                      *)
                        (* Calls RELBITREP, FLOAT, SAMERELPRECISION, *)
                        (* ADDARGS.                                *)
                        (* Possible BUGS - ARG will need to be     *)
                        (* unpacked or masked in Standard Pascal to  *)
                        (* make the NOT work properly.             *)
  BEGIN
    RELBITREP(2);
    ARG.BIT[2,1,MSBITNO] := NOT ARG.BIT[2,1,MSBITNO];
    FLOAT(1);
    SAMERELPRECISION;
    ADDARGS(2);
  END; (* MINUSIR *)
```

```
                    (* * * * * * * * * *)

PROCEDURE MINUSRI;    (* Same operation as PLUSRI except that the  *)
                      (* sign of ARG[2] is changed before the      *)
                      (* addition is performed.                    *)
                      (* Uses ARG, MSBITNO.                        *)
                      (* Calls FLOAT, RELBITREP, SAMERELPRECISION, *)
                      (* ADDARGS.                                  *)
                      (* Possible BUGS - ARG will need to be       *)
                      (* unpacked or masked in Standard Pascal to  *)
                      (* make the NOT work properly.               *)
  BEGIN
    FLOAT(2);
    ARG.BIT[2,1,MSBITNO] := NOT ARG.BIT[2,1,MSBITNO];
    RELBITREP(1);
    SAMERELPRECISION;
    ADDARGS(2);
  END; (* MINUSRI *)

                    (* * * * * * * * * *)

PROCEDURE MINUSI;     (* Same as PLUSI except that it complements  *)
                      (* the integer.                              *)
                      (* Uses WORDS.                               *)
                      (* Calls INTBITREP, COMPLMENT.               *)
  BEGIN
    INTBITREP(1);
    COMPLMENT(1,WORDS[1]);
  END; (* MINUSI *)

                    (* * * * * * * * * *)

PROCEDURE MINUSR;     (* Same as PLUSR except that the sign is changed *)
                      (* Uses ARG, MSBITNO.                            *)
                      (* Calls RELBITREP.                              *)
                      (* Possible BUGS - ARG will need to be unpacked  *)
                      (* or masked in Standard Pascal to make the NOT  *)
                      (* work properly.                                *)
  BEGIN
    RELBITREP(1);
    ARG.BIT[1,1,MSBITNO] := NOT ARG.BIT[1,1,MSBITNO];
  END; (* MINUSR *)

                    (* * * * * * * * * *)

PROCEDURE MULTII;
  BEGIN
  END; (* MULTII *)

                    (* * * * * * * * * *)
```

```
PROCEDURE MULTRR;
  BEGIN
  END; (* MULTRR *)

                (* * * * * * * * * *)

PROCEDURE MULTIR;
  BEGIN
  END; (* MULTIR *)

                (* * * * * * * * * *)

PROCEDURE MULTRI;
  BEGIN
  END; (* MULTRI *)

                (* * * * * * * * * *)

PROCEDURE DIVII;
  BEGIN
  END; (* DIVII *)

                (* * * * * * * * * *)

PROCEDURE DIVRR;
  BEGIN
  END; (* DIVRR *)

                (* * * * * * * * * *)

PROCEDURE DIVIR;
  BEGIN
  END; (* DIVIR *)

                (* * * * * * * * * *)

PROCEDURE DIVRI;
  BEGIN
  END; (* DIVRI *)

                (* * * * * * * * * *)

PROCEDURE REMI;
  BEGIN
  END; (* REMI *)

                (* * * * * * * * * *)
```

```
PROCEDURE MODI;
  BEGIN
  END; (* MODI *)

                    (* * * * * * * * * *)

PROCEDURE POWERII;
  BEGIN
  END; (* POWERII *)

                    (* * * * * * * * * *)

PROCEDURE POWERRR;
  BEGIN
  END; (* POWERRR *)

                    (* * * * * * * * * *)

PROCEDURE POWERIR;
  BEGIN
  END; (* POWERIR *)

                    (* * * * * * * * * *)

PROCEDURE POWERRI;
  BEGIN
  END; (* POWERRI *)

                    (* * * * * * * * * *)

PROCEDURE SQRTI;
  BEGIN
  END; (* SQRTI *)

                    (* * * * * * * * * *)

PROCEDURE SQRTR;
  BEGIN
  END; (* SQRTR *)

                    (* * * * * * * * * *)

PROCEDURE EXPI;
  BEGIN
  END; (* EXPI *)

                    (* * * * * * * * * *)
```

```
PROCEDURE EXPR;
  BEGIN
  END; (* EXPR *)

                    (* * * * * * * * * *)

PROCEDURE LOGI;
  BEGIN
  END; (* LOGI *)

                    (* * * * * * * * * *)

PROCEDURE LOGR;
  BEGIN
  END; (* LOGR *)

                    (* * * * * * * * * *)

PROCEDURE LNI;
  BEGIN
  END; (* LNI *)

                    (* * * * * * * * * *)

PROCEDURE LNR;
  BEGIN
  END; (* LNR *)

                    (* * * * * * * * * *)

PROCEDURE SINI;
  BEGIN
  END; (* SINI *)

                    (* * * * * * * * * *)

PROCEDURE SINR;
  BEGIN
  END; (* SINR *)

                    (* * * * * * * * * *)

PROCEDURE COSI;
  BEGIN
  END; (* COSI *)

                    (* * * * * * * * * *)
```

```
PROCEDURE COSR;
  BEGIN
  END; (* COSR *)

                    (* * * * * * * * * *)

PROCEDURE TANI;
  BEGIN
  END; (* TANI *)

                    (* * * * * * * * * *)

PROCEDURE TANR;
  BEGIN
  END; (* TANR *)

                    (* * * * * * * * * *)

PROCEDURE ARCSINI;
  BEGIN
  END; (* ARCSINI *)

                    (* * * * * * * * * *)

PROCEDURE ARCSINR;
  BEGIN
  END; (* ARCSINR *)

                    (* * * * * * * * * *)

PROCEDURE ARCCOSI;
  BEGIN
  END; (* ARCCOSI *)

                     (* * * * * * * * * *)

PROCEDURE ARCCOSR;
  BEGIN
  END; (* ARCCOSR *)

                    (* * * * * * * * * *)

PROCEDURE ARCTANI;
  BEGIN
  END; (* ARCTANI *)

                    (* * * * * * * * * *)
```

```
PROCEDURE ARCTANR;
  BEGIN
  END; (* ARCTANR *)

                    (* * * * * * * * * * *)

PROCEDURE SINHI;
  BEGIN
  END; (* SINHI *)

                    (* * * * * * * * * * *)

PROCEDURE SINHR;
  BEGIN
  END; (* SINHR *)

                    (* * * * * * * * * * *)

PROCEDURE COSHI;
  BEGIN
  END; (* COSHI *)

                    (* * * * * * * * * * *)

PROCEDURE COSHR;
  BEGIN
  END; (* COSHR *)

                    (* * * * * * * * * * *)

PROCEDURE TANHI;
  BEGIN
  END; (* TANHI *)

                    (* * * * * * * * * * *)

PROCEDURE TANHR;
  BEGIN
  END; (* TANHR *)
```

```
(*****************************************************************************)
(*****************************************************************************)


(*****************************************************************************)
(*                                                                           *)
(*                        MAIN MATHPAK EXECUTION                             *)
(*                                                                           *)
(*****************************************************************************)


BEGIN

    INITMASKS;
    ERR := NOERROR;
    ERRCODE := 0;
    CASE FUNCCODE OF
        1: FLOAT(1);
        2: RELTOINT;
        3: INTSIZECHANGE;
        4: RELSIZECHANGE;
        5: ABSVALI;
        6: ABSVALR;
       21: PLUSII;
       22: PLUSRR;
       23: PLUSIR;
       24: PLUSRI;
       25: PLUSI;
       26: PLUSR;
       41: MINUSII;
       42: MINUSRR;
       43: MINUSIR;
       44: MINUSRI;
       45: MINUSI;
       46: MINUSR;
       61: MULTII;
       62: MULTRR;
       63: MULTIR;
       64: MULTRI;
       81: DIVII;
       82: DIVRR;
       83: DIVIR;
       84: DIVRI;
       85: REMI;
       95: MODI;
      101: POWERII;
      102: POWERRR;
      103: POWERIR;
      104: POWERRI;
      115: SQRTI;
      116: SQRTR;
      125: EXPI;
```

```
        126: EXPR;
        145: LOGI;
        146: LOGR;
        155: LNI;
        156: LNR;
        165: SINI;
        166: SINR;
        175: COSI;
        176: COSR;
        185: TANI;
        186: TANR;
        195: ARCSINI;
        196: ARCSINR;
        205: ARCCOSI;
        206: ARCCOSR;
        215: ARCTANI;
        216: ARCTANR;
        225: SINHI;
        226: SINHR;
        235: COSHI;
        236: COSHR;
        245: TANHI;
        246: TANHR;
      END; (* CASE *)
      PUSHARG(1);
      IF (ORD(ERR) > 0) THEN
          ERRCODE := ORD(ERR);

END; (* MATHPAK *)


(*****************************************************************************)
(*****************************************************************************)
(**                                                                         **)
(**                        E N D   M A T H P A K                            **)
(**                                                                         **)
(*****************************************************************************)
(*****************************************************************************)
```

APPENDIX D

IMPLMENT Listing

```
(*****************************************************************************)
(*****************************************************************************)
(**                                                                         **)
(**                    P A C K A G E   I M P L M E N T                      **)
(**                                                                         **)
(*****************************************************************************)
(*****************************************************************************)


PROCEDURE IMPLMENT(FNCTIONCODE: INTEGER; VAR STACKPOINTER, ERRORCODE: INTEGER);


(*****************************************************************************)
(*****************************************************************************)

(* This is a Pascal Procedure which is meant to correspond to an Ada Package.
The code is written in Pascal, but it is written as much like Ada as possible
so that it can be translated into Ada. *)

(* This Package is designed to be used in conjunction with Package MATHPAK.
It does business in essentially the same manner as MATHPAK.  It contains
those routines which may be done differently for different implementations
but can be done this way for any implementation.  These routines could
be written in some other language, such as machine language.  As a separate
package they allow for an increase in efficiency where the particular machine
instruction set or hardware capabilities can be exploited to that end.  If
an implementation creates its own version of IMPLMENT, that version must
provide the same interface with MATHPAK as this one. *)

(* These routines were developed on the Apple II, but when completed they
should be implementation independent.  They only require that the constants
be set properly for the environment, in the same manner as, and consistent
with the constants of MATHPAK.
```

```
(*****************************************************************************)
(*****************************************************************************)

CONST                       (* Constants must be set to correspond to the     *)
                            (* environment - and MATHPAK                      *)
    WORDLENGTH = 16;        (* Number of bits in a machine word used to       *)
                            (*                 perform integer operations     *)
    LSBITNUM = 0;           (* The number of the least significant bit of an  *)
                            (*                              integer word      *)
    MSBITNUM = 15;          (* The number of the most significant bit of an   *)
                            (*                              integer word      *)
    BITLESS = -1;           (* One less than the smallest legal bit number of *)
                            (*                              an integer word   *)
    BITMORE = 16;           (* One more than the largest legal bit number of  *)
                            (*                              an integer word   *)
    BYTESIZE = 255;         (* Largest number which can be held in a half word*)
    BITSPERBYTE = 8;        (* Number of bits in a half word                  *)
    UPSTAK = -1;            (* -1 if stack built from high memory (bottom) to *)
                            (* low memory (top) and +1 if stack built opposite*)
    DOWNSTAK = 1;           (* +1 if stack built from high memory (bottom) to *)
                            (* low memory (top) and -1 if stack built opposite*)
    MORSIGBIT = 1;          (* +1 if bits in a word are numbered from least   *)
                            (* significant (lowest number) to most significant*)
                            (* (highest number) and -1 if bits numbered       *)
                            (* opposite way                                   *)
    LESSIGBIT = -1;         (* -1 if bits in a word are numbered from least   *)
                            (* significant (lowest number) to most significant*)
                            (* (highest number) and +1 if bits numbered       *)
                            (* opposite way                                   *)
    LARGESTARG = 16;        (* Largest number of words permitted for a        *)
                            (* numerical argument for this implementation     *)
    MAXNUMWORDS = 17;       (* One more than LARGESTARG                       *)
```

```
TYPE
    MAXBITNUMBER = BITLESS..BITMORE;
    MAXWORDS = 0..MAXNUMWORDS;
    ARGUINDEX = 1..2;
    SIGNRESULT = (EITHER,SAME);
    ERRORMESSAGE = (NOERROR,CONSTRAINTERROR,SIGNIFICANCELOST,OVERFLOW,
        UNDERFLOW,DIVISIONBYZERO,STORAGEERROR);
    BITNUMBER = LSBITNUM..MSBITNUM;
    WORKWORDS = 1..LARGESTARG;
    BYTENUMBER = 0..1;
    BITE = 0..BYTESIZE;
    REPS = (E,F,G,H);
    BYTEARAY = PACKED ARRAY[BYTENUMBER] OF BITE;
    BITARAY = PACKED ARRAY [BITNUMBER] OF BOOLEAN;
    REGISTER = RECORD CASE REPS OF              (* These representations    *)
        E: (INT: INTEGER);                      (* permit IMPLMENT to       *)
        F: (BOOL: BOOLEAN);                     (* interpret the contents   *)
        G: (BYTE: BYTEARAY);                    (* of the "registers" and   *)
        H: (BIT: BITARAY);                      (*                          *)
        END;                                    (*                          *)
    INTARAY = ARRAY[WORKWORDS] OF INTEGER;
    BOOLARAY = ARRAY[WORKWORDS] OF BOOLEAN;
    BITEARAY = ARRAY[WORKWORDS] OF BYTEARAY;
    BITTARAY = ARRAY[WORKWORDS] OF BITARAY;
    WORKARRAY = RECORD CASE REPS OF             (* arithmetic working area  *)
        E: (INT: INTARAY);                      (* any way it wants.        *)
        F: (BOOL: BOOLARAY);                    (* This makes bit and       *)
        G: (BYTE: BITEARAY);                    (* byte manipulations       *)
        H: (BIT: BITTARAY);                     (* possible.                *)
        END;                                    (*                          *)
```

```
VAR
    NUMBIT,                     (* Indexes for bit                    *)
    BITNUM,                     (*                                   *)
    BITNOM: MAXBITNUMBER;       (*                                   *)
    BYTENUM: BYTENUMBER;        (*            half word              *)
    WORDNUM,                    (*                      word         *)
    NUMWORDS: MAXWORDS;         (*                                   *)
    ARGUNUM,                    (*                           argument*)
    EXPNUM: ARGUINDEX;          (*                                numbers*)
    BYTEMASK,                   (*          "Registers"              *)
    SIGNMASK,                   (*                                   *)
    SIGN,                       (*                                   *)
    ARITHREG: REGISTER;         (*                                   *)
    WORKAREA,                   (*          Work areas               *)
    EXPVALUE,                   (*                                   *)
    RESULT: WORKARRAY;          (*                                   *)
    CARRY: 0..1;                (*          Carry value              *)
    EXPLENGTH,                  (* Real                              *)
    EXPPOINTER,                 (*            number                 *)
    FRACLENGTH,                 (*                                   *)
    SHIFTNUM,                   (*                        information*)
    POINTER,                    (*       Stack                       *)
    PTR,                        (*                    pointers       *)
    KOUNT,                      (*     General                       *)
    SIZE: INTEGER;              (*               counters            *)
    SIGNINDICATOR: SIGNRESULT;  (* For testing sign of result        *)
    ERRORIND: ERRORMESSAGE;     (* Error message holder              *)
```

```
(*****************************************************************************)
(*****************************************************************************)


(*****************************************************************************)
(*                                                                           *)
(*                            SUPPORT ROUTINES                               *)
(*                                                                           *)
(*****************************************************************************)

    PROCEDURE SIGNINTADD;  (* Checks to see if the signs of the two numbers *)
                           (* to be added are the same.  If not, then       *)
                           (* SIGNINDICATOR is set to EITHER to indicate    *)
                           (* that the result can have either sign.  If so, *)
                           (* then SIGNINDICATOR is set to SAME to indicate *)
                           (* that the result must have the same sign as    *)
                           (* SIGN.                                         *)
                           (* Uses WORKAREA, S, STACKPOINTER, SIGN,         *)
                           (* SIGNMASK, NUMWORDS, MSBITNUM, SIGNINDICATOR,  *)
                           (* DOWNSTAK.                                     *)
                           (* Possible BUGS - WORKAREA will need to be up-  *)
                           (* packed or masked in Standard Pascal.          *)
      BEGIN
        WORKAREA.INT[1] := S[STACKPOINTER];
        SIGN.BOOL := WORKAREA.BOOL[1] AND SIGNMASK.BOOL;
        WORKAREA.INT[2] := S[STACKPOINTER+NUMWORDS*DOWNSTAK];
        IF (WORKAREA.BIT[1,MSBITNUM] <> WORKAREA.BIT[2,MSBITNUM]) THEN
            SIGNINDICATOR := EITHER
        ELSE
            SIGNINDICATOR := SAME;
      END; (* SIGNINTADD *)

                        (* * * * * * * * * *)
```

```
PROCEDURE ADDGENERAL;    (* Performs the general addition function on *)
                         (* two arguments.  Takes one word at a time  *)
                         (* from each argument, starting with the     *)
                         (* least significant word, and puts one word *)
                         (* in WORKAREA[1] and the other in           *)
                         (* WORKAREA[2].  Takes one byte at a         *)
                         (* time from each word, starting with the    *)
                         (* least significant byte of each word.  Adds*)
                         (* in the carry from previous bytes added    *)
                         (* using CARRY.  Places the result in the    *)
                         (* corresponding byte of the RESULT indexed  *)
                         (* by WORDNUM.  Sets CARRY for the next add, *)
                         (* if any.  Sets the STACKPOINTER to the     *)
                         (* location immediately following the        *)
                         (* arguments (effectively having popped the  *)
                         (* arguments from the stack).                *)
                         (* Uses BYTENUM, ARITHREG, BYTEMASK, RESULT, *)
                         (* WORKAREA, CARRY, WORDNUM, NUMWORDS,       *)
                         (* POINTER, STACKPOINTER, PTR, S, DOWNSTAK,  *)
                         (* UPSTAK, KOUNT, BITSPERBYTE, MORSIGBIT,    *)
                         (* LSBITNUM.                                 *)
                         (* Possible BUGS - ARITHREG and WORKAREA     *)
                         (* will need to be unpacked or masked in     *)
                         (* Standard Pascal.                          *)
  BEGIN
    CARRY := 0;
    WORDNUM := NUMWORDS;
    POINTER := STACKPOINTER + NUMWORDS * DOWNSTAK + UPSTAK;
    PTR := POINTER + NUMWORDS * DOWNSTAK;
    FOR KOUNT := 1 TO NUMWORDS DO BEGIN
        WORKAREA.INT[2] := S[POINTER];
        WORKAREA.INT[1] := S[PTR];
        FOR BYTENUM := 0 TO 1 DO BEGIN
            ARITHREG.BYTE[0] := WORKAREA.BYTE[1,BYTENUM];
            ARITHREG.BOOL := ARITHREG.BOOL AND BYTEMASK.BOOL;
            WORKAREA.BYTE[3,0] := WORKAREA.BYTE[2,BYTENUM];
            WORKAREA.BOOL[3] := WORKAREA.BOOL[3] AND BYTEMASK.BOOL;
            ARITHREG.INT := ARITHREG.INT + WORKAREA.INT[3] + CARRY;
            WORKAREA.BYTE[4,BYTENUM] := ARITHREG.BYTE[0];
            IF (BYTENUM = 0) THEN BEGIN
                RESULT.BOOL[WORDNUM] := WORKAREA.BOOL[4] AND BYTEMASK.BOOL;
            END (* IF *)
            ELSE BEGIN
                WORKAREA.BOOL[4] := WORKAREA.BOOL[4] AND NOT BYTEMASK.BOOL;
                RESULT.BOOL[WORDNUM] := RESULT.BOOL[WORDNUM] OR
                    WORKAREA.BOOL[4];
            END; (* ELSE *)
            IF (ARITHREG.BIT[LSBITNUM+BITSPERBYTE*MORSIGBIT] = TRUE)
                THEN BEGIN
                CARRY := 1;
                ARITHREG.BIT[LSBITNUM+BITSPERBYTE*MORSIGBIT] := FALSE;
            END (* IF *)
            ELSE
```

```
            CARRY := 0;
        END; (* FOR *)
        WORDNUM := WORDNUM - 1;
        POINTER := POINTER + UPSTAK;
        PTR := PTR + UPSTAK;
    END; (* FOR *)
    STACKPOINTER := STACKPOINTER + (2 * NUMWORDS * DOWNSTAK);
END; (* ADDGENERAL *)

                    (* * * * * * * * * *)

PROCEDURE INTPREPADD;  (* Performs the general functions required  *)
                       (* for the addition of two integers.  Pops  *)
                       (* a word containing the number of          *)
                       (* words in each argument from the top of   *)
                       (* the stack.  Then calls SIGNINTADD to     *)
                       (* check for the sign to expect in the      *)
                       (* result, and calls ADDGENERAL to perform  *)
                       (* the addition.                            *)
                       (* Uses NUMWORDS, S, STACKPOINTER, DOWNSTAK.*)
                       (* Calls SIGNINTADD, ADDGENERAL.            *)
  BEGIN
    NUMWORDS := S[STACKPOINTER];
    STACKPOINTER := STACKPOINTER + DOWNSTAK;
    SIGNINTADD;
    ADDGENERAL;
  END; (* INTPREPADD *)

                    (* * * * * * * * * *)

PROCEDURE CHKHIWRDEND(VAR NUMOFWORD:  (* Increments the value of the *)
    MAXWORDS; VAR NUMOFBIT:           (* bit index, NUMOFBIT, and    *)
    MAXBITNUMBER);                    (* then checks to see if its   *)
                                      (* value is past a word boun-  *)
                                      (* dary.  If so, it decrements *)
                                      (* the word number, NUMOFWORD, *)
                                      (* and sets NUMOFBIT back to   *)
                                      (* the value of the least sig- *)
                                      (* nificant bit of a word,     *)
                                      (* LSBITNUM.                   *)
                                      (* Uses MSBITNUM, LSBITNUM,    *)
                                      (* MORSIGBIT.                  *)
  BEGIN
    NUMOFBIT := NUMOFBIT + MORSIGBIT;
    IF (((NUMOFBIT > MSBITNUM) AND (MORSIGBIT = 1)) OR
        ((NUMOFBIT < MSBITNUM) AND (MORSIGBIT = -1))) THEN BEGIN
        NUMOFBIT := LSBITNUM;
        NUMOFWORD := NUMOFWORD - 1;
    END; (* IF *)
  END; (* CHKHIWRDEND *)

                    (* * * * * * * * * *)
```

```
PROCEDURE CHKLOWRDEND(VAR NUMOFWORD:  (* Decrements the value of the *)
    MAXWORDS; VAR NUMOFBIT:           (* bit index, NUMOFBIT, and    *)
    MAXBITNUMBER);                    (* then checks to see if its   *)
                                      (* value is past a word boun-  *)
                                      (* dary.  If so, it increments *)
                                      (* the word number, NUMOFWORD, *)
                                      (* and sets NUMOFBIT back to   *)
                                      (* the value of the most sig-  *)
                                      (* nificant bit of a word,     *)
                                      (* MSBITNUM.                   *)
                                      (* Uses LSBITNUM, MSBITNUM,    *)
                                      (* LESSIGBIT.                  *)
  BEGIN
    NUMOFBIT := NUMOFBIT + LESSIGBIT;
    IF (((NUMOFBIT < LSBITNUM) AND (LESSIGBIT = -1)) OR
        ((NUMOFBIT > LSBITNUM) AND (LESSIGBIT = 1))) THEN BEGIN
        NUMOFBIT := MSBITNUM;
        NUMOFWORD := NUMOFWORD + 1;
    END; (* IF *)
  END; (* CHKLOWRDEND *)

                (* * * * * * * * * *)

PROCEDURE RIGHTSHIFT(OLDWORD:  (* Writes the bit in WORKAREA indexed *)
    MAXWORDS; OLDBIT:          (* by OLDWORD and OLDBIT to NEWWORD   *)
    MAXBITNUMBER; VAR NEWWORD: (* and NEWBIT.  OLDWORD and OLDBIT    *)
    MAXWORDS; VAR NEWBIT:      (* must begin at the index value for  *)
    MAXBITNUMBER);             (* the least significant bit to be    *)
                               (* shifted, and NEWWORD and NEWBIT    *)
                               (* begin at the index values for the  *)
                               (* least significant destination bit. *)
                               (* Works from the least significant   *)
                               (* bit up and shifts the number of    *)
                               (* bits specified by SHIFTNUM.        *)
                               (* Changes index values as it goes and*)
                               (* returns the final value of NEWWORD *)
                               (* and NEWBIT.                        *)
                               (* Uses SHIFTNUM, WORKAREA.           *)
                               (* Calls CHKHIWRDEND.                 *)
                               (* Possible BUGS - WORKAREA will need *)
                               (* to be unpacked or masked in        *)
                               (* Standard Pascal.                   *)
  BEGIN
    WHILE (SHIFTNUM > 0) DO BEGIN
        WORKAREA.BIT[NEWWORD,NEWBIT] := WORKAREA.BIT[OLDWORD,OLDBIT];
        SHIFTNUM := SHIFTNUM - 1;
        CHKHIWRDEND(OLDWORD,OLDBIT);
        CHKHIWRDEND(NEWWORD,NEWBIT);
    END; (* WHILE *)
  END; (* RIGHTSHIFT *)

                (* * * * * * * * * *)
```

```
PROCEDURE LEFTSHIFT(VAR NEWWORD:   (* Writes the bit in RESULT indexed *)
    MAXWORDS; VAR NEWBIT:          (* by OLDWORD and OLDBIT to NEWWORD *)
    MAXBITNUMBER; OLDWORD:         (* and NEWBIT.  OLDWORD and OLDBIT  *)
    MAXWORDS; OLDBIT:              (* must begin at the index value for*)
    MAXBITNUMBER);                 (* the most significant bit to be   *)
                                   (* shifted, and NEWWORD and NEWBIT  *)
                                   (* begin at the index values for the*)
                                   (* most significant destination bit.*)
                                   (* Works from the most significant  *)
                                   (* bit down and shifts the number of*)
                                   (* bits specified by SHIFTNUM.      *)
                                   (* Changes index values as it goes  *)
                                   (* and returns the final value of   *)
                                   (* NEWWORD and NEWBIT.              *)
                                   (* Uses SHIFTNUM, RESULT.           *)
                                   (* Calls CHKLOWRDEND.               *)
                                   (* Possible BUGS - RESULT will need *)
                                   (* to be unpacked or masked in      *)
                                   (* Standard Pascal.                 *)
  BEGIN
    WHILE (SHIFTNUM > 0) DO BEGIN
        RESULT.BIT[NEWWORD,NEWBIT] := RESULT.BIT[OLDWORD,OLDBIT];
        SHIFTNUM := SHIFTNUM - 1;
        CHKLOWRDEND(OLDWORD,OLDBIT);
        CHKLOWRDEND(NEWWORD,NEWBIT);
    END; (* WHILE *)
  END; (* LEFTSHIFT *)

                (* * * * * * * * * *)
```

```
PROCEDURE EXPFROMNUMBER(ARGUNO:(* Takes the first word of the argument *)
    ARGUINDEX);                (* from the stack and writes it to      *)
                               (* WORKAREA.  Then extracts the exponent*)
                               (* from it, writing the exponent into   *)
                               (* the appropriate word of EXPVALUE.    *)
                               (* Also extracts the sign bit and puts  *)
                               (* it in the sign bit of the EXPVALUE   *)
                               (* indexed 2 greater than the argument. *)
                               (* Uses EXPVALUE, EXPLENGTH, BITNUM,    *)
                               (* MSBITNUM, WORKAREA, S, EXPPOINTER,   *)
                               (* SIGNMASK, MORSIGBIT, LSBITNUM,       *)
                               (* LESSIGBIT.                           *)
                               (* Possible BUGS - EXPVALUE and WORKAREA*)
                               (* will need to be unpacked or masked in*)
                               (* Standard Pascal.                     *)
                               (* Known BUGS - assumes exponent length *)
                               (* of no more than 1 bit less than      *)
                               (* WORDLENGTH.                          *)
  VAR
    EXPBITNUM, WORKBITNUM: MAXBITNUMBER;
  BEGIN
    EXPVALUE.INT[ARGUNO] := 0;
    EXPBITNUM := EXPLENGTH * MORSIGBIT + LSBITNUM + LESSIGBIT;
    WORKBITNUM := MSBITNUM + LESSIGBIT;
    WORKAREA.INT[1] := S[EXPPOINTER];
    FOR BITNUM := 1 TO EXPLENGTH DO BEGIN
        EXPVALUE.BIT[ARGUNO,EXPBITNUM] := WORKAREA.BIT[1,WORKBITNUM];
        EXPBITNUM := EXPBITNUM + LESSIGBIT;
        WORKBITNUM := WORKBITNUM + LESSIGBIT;
    END; (* FOR *)
    EXPVALUE.BOOL[ARGUNO+2] := WORKAREA.BOOL[1] AND SIGNMASK.BOOL;
  END; (* EXPFROMNUMBER *)

                    (* * * * * * * * * * *)
```

```
FUNCTION EXPSIZE(WORDS:INTEGER): (* Calculates the proper sized exponent*)
    INTEGER;                     (* for a real argument from the number *)
                                 (* of words in the argument given as   *)
                                 (* the calling parameter.  It assumes  *)
                                 (* that the smallest possible word size*)
                                 (* is 8 bits.  It uses an algorithm    *)
                                 (* which determines the smallest expo- *)
                                 (* nent length which can hold an expo- *)
                                 (* nent equal to from +4 to -4 times   *)
                                 (* the length of the fractional part of*)
                                 (* the real number in bits.  It returns*)
                                 (* the number of bits in the length of *)
                                 (* this smallest exponent as the value *)
                                 (* of the function.                    *)
                                 (* Uses WORDLENGTH.                    *)
  VAR
    TRIALEXPO, TRIALFRAC, TRIALPRECISION, PRECIZION: INTEGER;
  BEGIN
    PRECIZION := WORDS * WORDLENGTH;
    TRIALEXPO := 5;
    TRIALFRAC := 4;
    TRIALPRECISION := TRIALEXPO + TRIALFRAC + 1;
    WHILE (TRIALPRECISION < PRECIZION) DO BEGIN
        TRIALEXPO := TRIALEXPO + 1;
        TRIALFRAC := TRIALFRAC * 2;
        TRIALPRECISION := TRIALEXPO + TRIALFRAC + 1;
    END; (* WHILE *)
    EXPSIZE := TRIALEXPO;
  END; (* EXPSIZE *)

                (* * * * * * * * * *)

PROCEDURE ARGTOWORKAREA(STACKWORD:  (* Writes the argument on the *)
    INTEGER);                       (* stack beginning at         *)
                                    (* S[STACKWORD] to WORKAREA.  *)
                                    (* Uses WORDNUM, NUMWORDS,    *)
                                    (* WORKAREA, S, DOWNSTAK.     *)
  BEGIN
    FOR WORDNUM := 1 TO NUMWORDS DO BEGIN
        WORKAREA.INT[WORDNUM] := S[STACKWORD];
        STACKWORD := STACKWORD + DOWNSTAK;
    END; (* FOR *)
  END; (* ARGTOWORKAREA *)

              (* * * * * * * * * *)
```

```
PROCEDURE WRKAREATOSTACK(STACKWORD: (* Takes an argument from      *)
    INTEGER);                       (* WORKAREA and puts it on the *)
                                    (* stack beginning at          *)
                                    (* S[STACKWORD].               *)
                                    (* Uses WORDNUM, NUMWORDS, S,  *)
                                    (* WORKAREA, DOWNSTAK.         *)
  BEGIN
    FOR WORDNUM := 1 TO NUMWORDS DO BEGIN
        S[STACKWORD] := WORKAREA.INT[WORDNUM];
        STACKWORD := STACKWORD + DOWNSTAK;
    END; (* FOR *)
  END; (* WRKAREATOSTACK *)

                (* * * * * * * * * *)

PROCEDURE ZEROOUTEXP(WORDOFSTACK:  (* Zeroes out the sign and exponent *)
    INTEGER);                      (* of the argument on the stack     *)
                                   (* starting at S[WORDOFSTACK].      *)
                                   (* Uses ARITHREG, S, BITNOM,        *)
                                   (* MSBITNUM, WORDLENGTH, EXPLENGTH, *)
                                   (* KOUNT, LESSIGBIT.                *)
                                   (* Possible BUGS - ARITHREG will    *)
                                   (* need to be unpacked or masked in *)
                                   (* Standard Pascal.                 *)
                                   (* Known BUGS - assumes the exponent*)
                                   (* to be no longer than 1 bit less  *)
                                   (* than WORDLENGTH.                 *)
  BEGIN
    ARITHREG.INT := S[WORDOFSTACK];
    BITNOM := MSBITNUM;
    FOR KOUNT := 0 TO EXPLENGTH DO BEGIN
        ARITHREG.BIT[BITNOM] := FALSE;
        BITNOM := BITNOM + LESSIGBIT;
    END; (* FOR *)
    S[WORDOFSTACK] := ARITHREG.INT;
  END; (* ZEROOUTEXP *)

                (* * * * * * * * * *)
```

```
PROCEDURE COMPLEMENT(STAKNUM:  (* Complements the argument on the stack *)
    INTEGER);                  (* beginning with S[STAKNUM].  Does not  *)
                               (* change a zero argument.               *)
                               (* Uses NUMWORDS, ARITHREG, S, WORDNUM,  *)
                               (* DOWNSTAK, UPSTAK.                     *)
  VAR
    CHANGEWORD,COUNT: INTEGER;
  BEGIN
    CHANGEWORD := STAKNUM + NUMWORDS * DOWNSTAK;
    COUNT := -1;
    REPEAT
        CHANGEWORD := CHANGEWORD + UPSTAK;
        COUNT := COUNT + 1;
        ARITHREG.INT := S[CHANGEWORD];
    UNTIL ((ARITHREG.INT <> 0) OR (CHANGEWORD = STAKNUM));
    IF (ARITHREG.INT <> 0)  THEN BEGIN
        FOR WORDNUM := (NUMWORDS-COUNT) DOWNTO 1 DO BEGIN
            ARITHREG.INT := S[STAKNUM+WORDNUM*DOWNSTAK+UPSTAK];
            ARITHREG.BOOL := NOT ARITHREG.BOOL;
            S[STAKNUM+WORDNUM*DOWNSTAK+UPSTAK] := ARITHREG.INT;
        END; (* FOR *)
        S[CHANGEWORD] := S[CHANGEWORD] + 1;
    END; (* IF *)
  END; (* COMPLEMENT *)

              (* * * * * * * * * *)

PROCEDURE EXPTONUMBER;     (* Puts the sign bit from SIGN into RESULT. *)
                           (* Then copies the exponent value from      *)
                           (* EXPVALUE[1] into RESULT.                 *)
                           (* Uses RESULT, MSBITNUM, SIGN, EXPLENGTH,  *)
                           (* BITNUM, EXPVALUE, MORSIGBIT, LSBITNUM,   *)
                           (* LESSIGBIT.                               *)
                           (* Possible BUGS - RESULT, SIGN, and        *)
                           (* EXPVALUE will need to be unpacked or     *)
                           (* masked in Standard Pascal.               *)
  VAR
    EXPBIT, RESBIT: MAXBITNUMBER;
  BEGIN
    RESULT.BIT[1,MSBITNUM] := SIGN.BIT[MSBITNUM];
    EXPBIT := EXPLENGTH * MORSIGBIT + LSBITNUM + LESSIGBIT;
    RESBIT := MSBITNUM + LESSIGBIT;
    FOR BITNUM := 1 TO EXPLENGTH DO BEGIN
        RESULT.BIT[1,RESBIT] := EXPVALUE.BIT[1,EXPBIT];
        EXPBIT := EXPBIT + LESSIGBIT;
        RESBIT := RESBIT + LESSIGBIT;
    END; (* FOR *)
  END; (* EXPTONUMBER *)

              (* * * * * * * * * *)
```

```
PROCEDURE DECEXP(VAR UNDFLOW:   (* If UNDFLOW already set, it does      *)
    BOOLEAN);                   (* nothing.  If not, it decrements the  *)
                                (* exponent value by 1 and then checks  *)
                                (* for exponent underflow (negative     *)
                                (* value).  If found, it restores zero  *)
                                (* to the exponent, sets UNDFLOW, and   *)
                                (* sets SHIFTNUM back to maximum allow- *)
                                (* able shift.  If no underflow, it     *)
                                (* increments SHIFTNUM.                 *)
                                (* Uses EXPVALUE, SHIFTNUM.             *)
                                (* Known BUGS - assumes the exponent to *)
                                (* be no longer than one word.          *)
  BEGIN
    IF (UNDFLOW = FALSE) THEN BEGIN
        EXPVALUE.INT[1] := EXPVALUE.INT[1] - 1;
        IF (EXPVALUE.INT[1] < 0) THEN BEGIN
            UNDFLOW := TRUE;
            EXPVALUE.INT[1] := 0;
            SHIFTNUM := SHIFTNUM - 1;
        END (* IF *)
        ELSE
            SHIFTNUM := SHIFTNUM + 1;
    END; (* IF *)
  END; (* DECEXP *)

                (* * * * * * * * * * *)
```

```
PROCEDURE CARRYONMSB;    (* Checks for exponent overflow and returns *)
                         (* an error if found.  Otherwise increases  *)
                         (* the exponent by 1 and shifts the argument*)
                         (* in RESULT by 1 to the right.             *)
                         (* Uses EXPVALUE, ERRORIND, WORDNUM,        *)
                         (* NUMWORDS, BITNUM, LSBITNUM, WORKAREA,    *)
                         (* RESULT, MORSIGBIT.                       *)
                         (* Calls RIGHTSHIFT.                        *)
                         (* Known BUGS - assumes exponent is no more *)
                         (* than 1 bit less than WORDLENGTH.         *)
  VAR
    WORDCNT: MAXWORDS;
  BEGIN
    EXPVALUE.INT[1] := EXPVALUE.INT[1] + 1;
    IF (EXPVALUE.INT[1] < 0) THEN BEGIN
        EXPVALUE.INT[1] := EXPVALUE.INT[2];
        ERRORIND := OVERFLOW;
    END (* IF *)
    ELSE BEGIN
        WORDNUM := NUMWORDS;
        BITNUM := LSBITNUM;
        FOR WORDCNT := 1 TO NUMWORDS DO
            WORKAREA.INT[WORDCNT] := RESULT.INT[WORDCNT];
        RIGHTSHIFT(NUMWORDS,LSBITNUM+MORSIGBIT,WORDNUM,BITNUM);
        FOR WORDCNT := 1 TO NUMWORDS DO
            RESULT.INT[WORDCNT] := WORKAREA.INT[WORDCNT];
    END; (* ELSE *)
  END; (* CARRYONMSB *)

                  (* * * * * * * * * * *)
```

```
PROCEDURE NORMLEFTSHIFT;                (* Checks for first non zero bit *)
                                        (* to determine how far to shift *)
                                        (* for normalization.  If all    *)
                                        (* zeroes are found, then expo-  *)
                                        (* nent and sign are also zeroed *)
                                        (* out.  If exponent underflow   *)
                                        (* occurs, an UNDERFLOW error is *)
                                        (* returned and the number       *)
                                        (* shifted as far as possible.   *)
                                        (* Otherwise, the complete nor-  *)
                                        (* alization shift is made and   *)
                                        (* the exponent is decremented   *)
                                        (* accordingly.                  *)
                                        (* Uses WORDNUM, BITNUM, RESULT, *)
                                        (* NUMWORDS, LSBITNUM, EXPVALUE, *)
                                        (* SIGN, WORDLENGTH, EXPLENGTH,  *)
                                        (* FRACLENGTH, SHIFTNUM,         *)
                                        (* ERRORIND, MSBITNUM, LESSIGBIT.*)
                                        (* Calls CHKLOWRDEND, DECEXP,    *)
                                        (* LEFTSHIFT.                    *)
                                        (* Possible BUGS - RESULT will   *)
                                        (* need to be unpacked or masked *)
                                        (* in Standard Pascal.           *)
  VAR
    UNDRFLOW: BOOLEAN;
    WORDCNT: MAXWORDS;
    BITCNT: MAXBITNUMBER;
  BEGIN
    UNDRFLOW := FALSE;
    REPEAT
        CHKLOWRDEND(WORDNUM,BITNUM);
        DECEXP(UNDRFLOW);
    UNTIL ((RESULT.BIT[WORDNUM,BITNUM] = TRUE) OR ((WORDNUM = NUMWORDS) AND
        (BITNUM = LSBITNUM)));
    IF (RESULT.BIT[WORDNUM,BITNUM] = FALSE) THEN BEGIN
        UNDRFLOW := FALSE;
        EXPVALUE.INT[1] := 0;
        SIGN.INT := 0;
    END (* IF *)
    ELSE BEGIN
        WORDCNT := 1;
        BITCNT := MSBITNUM + (EXPLENGTH + 1) * LESSIGBIT;
        SHIFTNUM := FRACLENGTH - SHIFTNUM;
        LEFTSHIFT(WORDCNT,BITCNT,WORDNUM,BITNUM);
        REPEAT
            RESULT.BIT[WORDCNT,BITCNT] := FALSE;
            CHKLOWRDEND(WORDCNT,BITCNT);
        UNTIL (WORDCNT > NUMWORDS);
    END; (* ELSE *)
    IF (UNDRFLOW = TRUE) THEN
        ERRORIND := UNDERFLOW;
  END; (* NORMLEFTSHIFT *)
```

```
                    (* * * * * * * * * *)

PROCEDURE ADDNORMALIZE;     (* Checks for carry on the most significant  *)
                            (* bit, and if found calls CARRYONMSB for    *)
                            (* right shift.  Otherwise checks for        *)
                            (* possible left shift for normalization,    *)
                            (* and if necessary, calls NORMLEFTSHIFT.    *)
                            (* Uses WORDNUM, BITNUM, WORDLENGTH,         *)
                            (* EXPLENGTH, RESULT, SHIFTNUM, FRACLENGTH,  *)
                            (* MSBITNUM, LESSIGBIT.                      *)
                            (* Calls CARRYONMSB, CHKLOWRDEND,            *)
                            (* NORMLEFTSHIFT.                            *)
                            (* Possible BUGS - RESULT will need to be    *)
                            (* unpacked or masked in Standard Pascal.    *)
  BEGIN
    WORDNUM := 1;
    BITNUM := MSBITNUM + EXPLENGTH * LESSIGBIT;
    IF (RESULT.BIT[1,BITNUM] = TRUE) THEN BEGIN
        SHIFTNUM := FRACLENGTH;
        CARRYONMSB;
    END (* IF *)
    ELSE BEGIN
        SHIFTNUM := 0;
        CHKLOWRDEND(WORDNUM,BITNUM);
        IF (RESULT.BIT[WORDNUM,BITNUM] = FALSE) THEN
            NORMLEFTSHIFT;
    END; (* ELSE *)
  END; (* ADDNORMALIZE *)

                    (* * * * * * * * * *)
```

```
PROCEDURE RELFRACADDPREP; (* Shifts the fractional part of the smaller *)
                          (* number, if necessary, by the amount indi- *)
                          (* cated by SHIFTNUM.  Zeroes out the sign   *)
                          (* and exponent of both arguments.  Then     *)
                          (* complements either fractional part if it  *)
                          (* represents a negative number, and finally *)
                          (* adds the fractional parts, with the result*)
                          (* ending up in RESULT.                      *)
                          (* Uses SHIFTNUM, EXPNUM, STACKPOINTER,      *)
                          (* NUMWORDS, FRACLENGTH, WORDLENGTH,         *)
                          (* EXPLENGTH, WORDNUM, BITNOM, LSBITNUM,     *)
                          (* MSBITNUM, POINTER, ARGUNUM, EXPVALUE,     *)
                          (* WORKAREA, DOWNSTAK, MORSIGBIT, LESSIGBIT. *)
                          (* Calls ARGTOWORKAREA, RIGHTSHIFT,          *)
                          (* WRKAREATOSTACK, COMPLEMENT, SIGNINTADD,   *)
                          (* ADDGENERAL, CHKLOWRDEND, ZEROOUTEXP.      *)
                          (* Possible BUGS - WORKAREA will need to be  *)
                          (* unpacked or masked in Standard Pascal.    *)
                          (* Known BUGS - assumes the exponent is no   *)
                          (* longer than 1 bit less than WORDLENGTH.   *)
  VAR
    FRACWORD: MAXWORDS;
    FRACBIT,
    STAKWORD: INTEGER;
  BEGIN
    FRACLENGTH := (NUMWORDS * WORDLENGTH) - EXPLENGTH - 1;
    IF (SHIFTNUM <> 0) THEN BEGIN
        IF (EXPNUM = 2) THEN
            STAKWORD := STACKPOINTER
        ELSE
            STAKWORD := STACKPOINTER + NUMWORDS * DOWNSTAK;
        ARGTOWORKAREA(STAKWORD);
        FRACBIT := SHIFTNUM;
        FRACWORD := NUMWORDS;
        WHILE (FRACBIT > WORDLENGTH) DO BEGIN
            FRACWORD := FRACWORD - 1;
            FRACBIT := FRACBIT - WORDLENGTH;
        END; (* WHILE *)
        FRACBIT := LSBITNUM + FRACBIT * MORSIGBIT;
        SHIFTNUM := FRACLENGTH - SHIFTNUM;
        WORDNUM := NUMWORDS;
        BITNOM := LSBITNUM;
        RIGHTSHIFT(FRACWORD,FRACBIT,WORDNUM,BITNOM);
        FRACWORD := 1;
        FRACBIT := MSBITNUM + EXPLENGTH * LESSIGBIT;
        REPEAT
            CHKLOWRDEND(FRACWORD,FRACBIT);
            WORKAREA.BIT[FRACWORD,FRACBIT] := FALSE;
        UNTIL ((FRACWORD = WORDNUM) AND (FRACBIT = BITNOM));
        WRKAREATOSTACK(STAKWORD);
    END; (* IF *)
    ZEROOUTEXP(STACKPOINTER);
```

```
  ZEROOUTEXP(STACKPOINTER+NUMWORDS*DOWNSTAK);
  POINTER := STACKPOINTER;
  FOR ARGUNUM := 2 DOWNTO 1 DO BEGIN
      IF (EXPVALUE.INT[ARGUNUM+2] < 0) THEN
          COMPLEMENT(POINTER);
      POINTER := STACKPOINTER + NUMWORDS * DOWNSTAK;
  END; (* FOR *)
  SIGNINTADD;
  ADDGENERAL;
END; (* RELFRACADDPREP *)

                    (* * * * * * * * * * *)
```

```
PROCEDURE RELEXPADDPREP;  (* Reads in the value of the number of words *)
                          (* in each argument from the top of the stack*)
                          (* and determines the length of the exponents*)
                          (* from it.  Reads each exponent into       *)
                          (* EXPVALUE and then checks to see which, if *)
                          (* either, is the larger.  Sets SHIFTNUM to  *)
                          (* the difference between the exponent       *)
                          (* values.  If different exponent values,    *)
                          (* sets EXPNUM to the index of the smaller   *)
                          (* exponent (indicating which argument needs *)
                          (* to be shifted) and then makes both expo-  *)
                          (* nents equal to the larger value.  Checks  *)
                          (* for possible underflow if all significant *)
                          (* bits would be shifted out of the number,  *)
                          (* and returns an underflow error and sets   *)
                          (* the argument equal to 0 in this case.     *)
                          (* Also checks bits to be lost in shifting   *)
                          (* and returns a significance warning if any *)
                          (* are 1's.                                  *)
                          (* Uses NUMWORDS, S, STACKPOINTER, EXPLENGTH,*)
                          (* EXPPOINTER, EXPVALUE, SHIFTNUM, EXPNUM,   *)
                          (* POINTER, FRACLENGTH, WORDLENGTH, ERRORIND,*)
                          (* WORDNUM, BITNUM, LSBITNUM, WORKAREA,      *)
                          (* MSBITNUM, DOWNSTAK, KOUNT, UPSTAK,        *)
                          (* MORSIGBIT.                                *)
                          (* Calls EXPSIZE, EXPFROMNUMBER.             *)
                          (* Possible BUGS - WORKAREA will need to be  *)
                          (* unpacked or masked in Standard Pascal.    *)
                          (* Known BUGS - uses only exponents no longer*)
                          (* than 1 bit less than WORDLENGTH.          *)
  VAR
    CHECKNUM: INTEGER;
    CHECKVAL: BOOLEAN;
  BEGIN
    NUMWORDS := S[STACKPOINTER];
    STACKPOINTER := STACKPOINTER + DOWNSTAK;
    EXPLENGTH := EXPSIZE(NUMWORDS);
    EXPPOINTER := STACKPOINTER;
    EXPFROMNUMBER(2);
    EXPPOINTER := STACKPOINTER + NUMWORDS * DOWNSTAK;
    EXPFROMNUMBER(1);
    IF (EXPVALUE.INT[1] = EXPVALUE.INT[2]) THEN
        SHIFTNUM := 0
    ELSE BEGIN
        IF (EXPVALUE.INT[1] > EXPVALUE.INT[2]) THEN BEGIN
            EXPNUM := 2;
            SHIFTNUM := EXPVALUE.INT[1] - EXPVALUE.INT[2];
            POINTER := STACKPOINTER;
            EXPVALUE.INT[2] := EXPVALUE.INT[1];
        END (* IF *)
        ELSE BEGIN
            EXPNUM := 1;
            SHIFTNUM := EXPVALUE.INT[2] - EXPVALUE.INT[1];
```

```
            POINTER := STACKPOINTER + NUMWORDS * DOWNSTAK;
            EXPVALUE.INT[1] := EXPVALUE.INT[2];
        END; (* ELSE *)
        FRACLENGTH := (NUMWORDS * WORDLENGTH) - EXPLENGTH - 1;
        IF (SHIFTNUM >= FRACLENGTH) THEN BEGIN
            SHIFTNUM := 0;
            ERRORIND := UNDERFLOW;
            WORDNUM := POINTER;
            FOR KOUNT := 1 TO NUMWORDS DO BEGIN
                S[WORDNUM] := 0;
                WORDNUM := WORDNUM + DOWNSTAK;
            END; (* FOR *)
        END (* IF *)
        ELSE BEGIN
            POINTER := POINTER + NUMWORDS * DOWNSTAK;
            BITNOM := LSBITNUM;
            CHECKNUM := SHIFTNUM;
            REPEAT
                POINTER := POINTER + UPSTAK;
                WORKAREA.INT[1] := S[POINTER];
                REPEAT
                    CHECKVAL := WORKAREA.BIT[1,BITNOM];
                    CHECKNUM := CHECKNUM - 1;
                    BITNOM := BITNOM + MORSIGBIT;
                UNTIL ((CHECKVAL = TRUE) OR (CHECKNUM = 0) OR
                    (((BITNOM > MSBITNUM) AND (MORSIGBIT = 1)) OR
                    ((BITNOM < MSBITNUM) AND (MORSIGBIT = -1))));
                BITNOM := LSBITNUM;
            UNTIL ((CHECKVAL = TRUE) OR (CHECKNUM = 0));
            IF (CHECKVAL = TRUE) THEN
                ERRORIND := SIGNIFICANCELOST;
        END; (* ELSE *)
    END; (* ELSE *)
END; (* RELEXPADDPREP *)

                    (* * * * * * * * * * *)

PROCEDURE PUSHRESULT; (* Takes the argument from RESULT and places     *)
                      (* it on the stack starting at the position above*)
                      (* that pointed to by STACKPOINTER.  Leaves      *)
                      (* STACKPOINTER pointing to the first word of the*)
                      (* argument (top of the stack).                  *)
                      (* Uses WORDNUM, NUMWORDS, STACKPOINTER, S,      *)
                      (* RESULT, UPSTAK.                               *)
BEGIN
    FOR WORDNUM := NUMWORDS DOWNTO 1 DO BEGIN
        STACKPOINTER := STACKPOINTER + UPSTAK;
        S[STACKPOINTER] := RESULT.INT[WORDNUM];
    END; (* FOR *)
END; (* PUSHRESULT *)

                    (* * * * * * * * * * *)
```

```
PROCEDURE CHKINTADD;  (* Checks SIGNINDICATOR to see if the result is *)
                      (* supposed to have the same sign as SIGN.  If  *)
                      (* so, it checks the signs and if they don't    *)
                      (* match, it sets ERRORIND to indicate an       *)
                      (* overflow and then sets the sign back to the  *)
                      (* correct value.                               *)
                      (* Uses SIGNINDICATOR, RESULT, MSBITNUM, SIGN,  *)
                      (* ERRORIND, SIGNMASK.                          *)
                      (* Possible BUGS - RESULT and SIGN will need to *)
                      (* be unpacked or masked in Standard Pascal.    *)
  BEGIN
    IF (SIGNINDICATOR = SAME) THEN BEGIN
        IF (RESULT.BIT[1,MSBITNUM] <> SIGN.BIT[MSBITNUM]) THEN BEGIN
            ERRORIND := OVERFLOW;
            RESULT.BOOL[1] := SIGN.BOOL OR (RESULT.BOOL[1] AND NOT
                SIGNMASK.BOOL);
        END; (* IF *)
    END; (* IF *)
  END; (* CHKINTADD *)

                    (* * * * * * * * * *)

PROCEDURE CHKRELADD;  (* Complements the resultant fractional part *)
                      (* if negative, then normalizes the frac-    *)
                      (* tional part and fills in the sign and     *)
                      (* exponent to complete the final result in  *)
                      (* RESULT.                                   *)
                      (* Uses SIGN, RESULT, MSBITNUM, STACKPOINTER,*)
                      (* NUMWORDS, WORDNUM, S, DOWNSTAK.           *)
                      (* Calls PUSHRESULT, COMPLEMENT, EXPTONUMBER,*)
                      (* ADDNORMALIZE.                             *)
                      (* Possible BUGS - SIGN and RESULT will need *)
                      (* to be unpacked or masked in Standard      *)
                      (* Pascal.                                   *)
  BEGIN
    SIGN.BIT[MSBITNUM] := RESULT.BIT[1,MSBITNUM];
    IF (RESULT.INT[1] < 0) THEN BEGIN
        PUSHRESULT;
        COMPLEMENT(STACKPOINTER);
        FOR WORDNUM := 1 TO NUMWORDS DO BEGIN
            RESULT.INT[WORDNUM] := S[STACKPOINTER];
            STACKPOINTER := STACKPOINTER + DOWNSTAK;
        END; (* FOR *)
    END; (* IF *)
    ADDNORMALIZE;
    EXPTONUMBER;
  END; (* CHKRELADD *)

                  (* * * * * * * * * *)
```

```
PROCEDURE MASKINIT;      (* Initializes BYTEMASK - byte 1 is all zeroes *)
                         (* and byte 0 is all ones.  Also initializes   *)
                         (* SIGNMASK with a 1 in the sign bit and the   *)
                         (* rest zeroes.                                *)
                         (* Uses BYTEMASK, SIGNMASK, BITNUM, WORDLENGTH,*)
                         (* BYTESIZE.                                   *)
  BEGIN
    BYTEMASK.INT := BYTESIZE;
    SIGNMASK.INT := 1;
    FOR BITNUM := 1 TO (WORDLENGTH - 1) DO
        SIGNMASK.INT := 2 * SIGNMASK.INT;
  END; (* MASKINIT *)
```

```
(*****************************************************************************)
(*****************************************************************************)


(*****************************************************************************)
(*                                                                           *)
(*                          IMPLMENT "FUNCTIONS"                             *)
(*                                                                           *)
(*****************************************************************************)


    PROCEDURE INTADD;    (* Adds two integers which are on the stack pre-  *)
                         (* ceded by an integer word which indicates the   *)
                         (* number of words in each integer argument.      *)
                         (* Places the result back on the top of the stack.*)
                         (* Uses no variables.                             *)
                         (* Calls INTPREPADD, CHKINTADD.                   *)
      BEGIN
        INTPREPADD;
        CHKINTADD;
      END; (* INTADD *)

                       (* * * * * * * * * *)

    PROCEDURE RELADD;    (* Adds two reals which are on the stack preceded *)
                         (* by an integer word which indicates the number  *)
                         (* of words in each real argument.  Places the    *)
                         (* result back on the top of the stack.           *)
                         (* Uses no variables.                             *)
                         (* Calls RELFRACADDPREP, RELEXPADDPREP,           *)
                         (* CHKRELADD.                                     *)
      BEGIN
        RELEXPADDPREP;
        RELFRACADDPREP;
        CHKRELADD;
      END; (* RELADD *)

                           (* * * * * * * * * *)

    PROCEDURE INTMULT;
      BEGIN
      END; (* INTMULT *)

                           (* * * * * * * * * *)

    PROCEDURE RELMULT;
      BEGIN
      END; (* RELMULT *)

                           (* * * * * * * * * *)
```

```
PROCEDURE INTDIV;
  BEGIN
  END; (* INTDIV *)

                        (* * * * * * * * * * *)

PROCEDURE RELDIV;
  BEGIN
  END; (* RELDIV *)
```

```
(*****************************************************************************)
(*****************************************************************************)


(*****************************************************************************)
(*                                                                           *)
(*                          MAIN IMPLMENT EXECUTION                          *)
(*                                                                           *)
(*****************************************************************************)


BEGIN

    MASKINIT;
    ERRORIND := NOERROR;
    CASE FNCTIONCODE OF
        1: INTADD;
        2: RELADD;
        3: INTMULT;
        4: RELMULT;
        5: INTDIV;
        6: RELDIV;
    END; (* CASE *)
    PUSHRESULT;
    IF (ORD(ERRORIND) > 0) THEN
        ERRORCODE := ORD(ERRORIND);

END; (* IMPLMENT *)


(*****************************************************************************
(*****************************************************************************)
(**                                                                         **)
(**                      E N D   I M P L M E N T                            **)
(**                                                                         **)
(*****************************************************************************)
(*****************************************************************************)
```

VITA

Patricia K. Lawlis was born on 5 May 1945 in Greensburg, Pennsylvania. She graduated from Charleroi High School in Charleroi, Pennsylvania in 1963. After receiving a Bachelor of Science Degree in Mathematics from East Carolina College in Greenville, North Carolina, in February of 1967, she taught high school mathematics until entering the Air Force in July of 1974. She was commissioned through OTS in October of the same year. She was then assigned to Colorado Springs, Colorado, first with the 1st Aerospace Control Squadron at Ent AFB and later (after Ent closed in 1975) with the 4602nd Computer Services Squadron at Peterson AFB. She worked as a space systems analyst on the 427M project for the NORAD Cheyenne Mountain Complex. In November of 1977 she was assigned to Detachment 1, 601st Tactical Control Group, and attached to the Neubruecke Army Installation in West Germany. Her duty was as a systems analyst in the joint US-German Programming Center Birkenfeld located on the Heinrich Hertz Kaserne, a German Air Base. She left Germany when assigned to the Air Force Institute of Technology School of Engineering at Wright-Patterson AFB, Ohio in August of 1980. She is a member of Tau Beta Pi.

Permanent address: 401 Lookout Avenue
Charleroi, Pennsylvania 15022

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/MA/82M-1	2. GOVT ACCESSION NO. AD-A115 557	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ARBITRARY PRECISION IN A PRELIMINARY MATH UNIT FOR ADA	5. TYPE OF REPORT & PERIOD COVERED MS Thesis
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Patricia K. Lawlis Capt USAF	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT-EN) Wright-Patterson AFB, Ohio 45433	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Avionics Laboratory (AFWAL/AAAF-3) Wright-Patterson AFB, Ohio 45433	12. REPORT DATE March, 1982
	13. NUMBER OF PAGES 136
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

15 APR 1982

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB, OH 45433

18. SUPPLEMENTARY NOTES

Approved for public release; IAW AFR 190-17

~~F. C. LYNCH, Major, USAF~~
Director of Information

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Mathematics	Floating Point
Computer Arithmetic	Transcendental Functions
Computer Languages	Ada
Pascal	Ada Language Environment

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This project involved designing a unit of mathematical functions for an Ada language environment. The design includes the basic arithmetic operations and the transcendental functions. It uses modularity and defines both integer and floating point number representations. The main features of the design are its use of any specified precision (referred to as arbitrary precision), its implementation independence, and its structure as two separate Ada-like packages. Two packages are used to permit an implementation to increase the unit's

(Continued on reverse)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

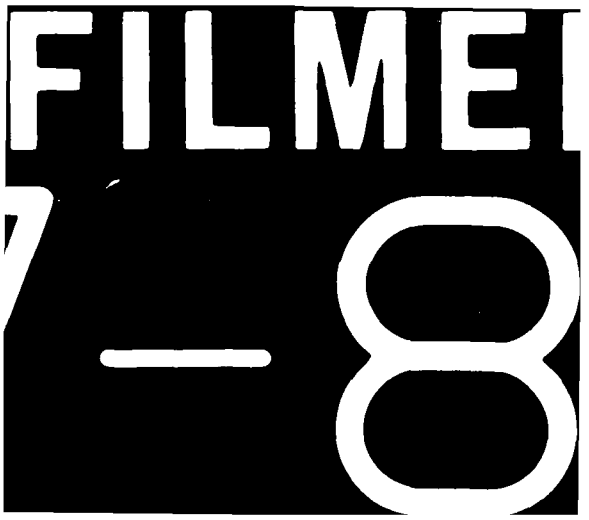
UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(Continued from 20)

efficiency, if desirable. The main package, MATHPAK, sets up all of the unit's functions a [illegible] the [illegible] level arithmetic operations. The secondary [illegible], [illegible], [illegible] the machine level arithmetic operations, and its implementation independent version can be replaced by a more efficient, machine dependent package without requiring changes in MATHPAK.

A partial implementation of the design was included as part of the project. This implementation was written in Pascal, since validated Ada environments did not yet exist. It is independent of any particular machine, and it implements arbitrary precision addition and subtraction for both integers and reals.


FILME
7 — 8